1984

Investigation and implementation of a tree transformation system for user friendly programming

Chok, Mohamed B.

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

INVESTIGATION AND IMPLEMENTATION
OF A TREE TRANSFORMATION SYSTEM
FOR USER FRIENDLY PROGRAMMING

by

Mohamed B. Chok

December 1984

Thesis Advisor: Bruce J. MacLennan

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Investigation and Implementation of a Tree Transformation System for User Friendly Programming		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mohamed B. Chok		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE December 1984
		13. NUMBER OF PAGES 136
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

template, concrete transformation rule, abstract transformation rule, abstract tree, tree transformation, tree pattern matching, interpreter, synthesization, term rewriting system, application programming, functional programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The programming system (TTPS) described in this thesis is based on tree transformation techniques, comonly known as abstract transformation. The objects manipulated by the user through "TTPS" are: the templates, the transformation rules, and the programs. The templates define the syntactic and semantic language framework which will be used to parse and unparse both the rules and the program trees. The rules define the semantic behavior of the transformation process. The program represents the (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

source tree which describes the problem to solve, and will be interpreted by a successive application of the supplied rules until they no longer apply.
"TTPS" provides an appropriate environment for a large class of applications (e.g. system programming, code generation, structure transformation, simulation of syntax directed editors, and other conventional applications), and supports many programming styles such as functional programming, conventional programming, and user defined style.

Approved for public release; distribution is unlimited.

Investigation and Implementation of a Tree Transformation System for User Friendly Programming

by

Mohamed B. Chok
Captain, Tunisian Army

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

ABSTRACT

The programming system (TTPS) described in this thesis is based on tree transformation techniques, commonly known as abstract transformation. The objects manipulated by the user through "TTPS" are: the templates, the transformation rules, and the programs. The templates define the syntactic and semantic language framework which will be used to parse and unparse both the rules and the program trees. The rules define the semantic behavior of the transformation process. The program represents the source tree which describes the problem to solve, and will be interpreted by a successive application of the supplied rules until they no longer apply.

"TTPS" provides an appropriate environment for a large class of applications (e.g. system programming, code generation, structure transformation, simulation of syntax directed editors, and other conventional applications), and supports many programming styles such as functional programming, conventional programming, and user defined style.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my sincere appreciations and gratitude to my thesis Advisor, Professor Bruce J. MacLennan, who suggested this tcpic, constantly contributed to the development of the thesis, and spent much of his time guiding my research and reading the drafts of the thesis.

Thanks go to my Second Reader, Professor Gordon H. Bradley, not only for the reading of the final draft, but also for his assistance provided to me to select the topic, and for his helpful criticism.

I wish to think all the personnel in the Computer Technology Curricular Office, and in the International Coordinator Education Office, for their constant interest, care and encouragement throughout the course.

Special thanks are due to my wife Hajer, and my children Hamed and Aymen, for their understanding, support, and patience.

I. INTRODUCTION

A. MOTIVATION, VIEWPOINT

Although high level languages permits the writing of programs in a form more convenient to human beings, programming is still a rather difficult task, which requires a lot of training. As a result, access to the computer remains more or less restricted to a class of trained personal, who can do the necessary coding and debugging.

Extensive research has been done to find new ways to make programming systems more friendly, more flexible, and easier to learn and use. Thus, during the last decade, we have witnessed the development of new classes of programming system such as: object oriented languages and applicative programming.

Along with this research, people are focusing more and more on user friendly man_machine interfaces and programming environments.

Man_machine interfaces can be viewed as tools, which enable the users to communicate with a computer in a friendly, flexible, and easy way, often using formatted natural language to present the information to the computer in order to make it perform specific tasks. This view holds that natural language interfaces might be appropriate for people who have a high level of semantic knowledge in a problem domain, but aren't familiar with, or are unwilling to learn, a formal computer language.

Programming environments are a collection of automated tools, which provide assistance to the programmer in the different steps of the program development process (i.e. the life cycle).

The modern programming environment has evolved from interpreters, compilers, and common operating system, to include more sophisticated and elaborate tools such as: syntax directed editors, structure editors, debuggers, automatic program generators, pretty printers, file system coordinators and others. Programming environment systems have become an important area of research because of their direct impact on all areas of computer science such as software engineering, programming languages, and artificial intelligence. Thus, programming systems are no longer evaluated by the language alone, but by the entire environment in which programs are developed.

In this thesis we will investigate and implement a friendly programming system based on tree transformation techniques, commonly known as abstract transformation. In fact any language structure can potentially be represented as an abstract syntax tree (e.g. expression, control statement, input/output statement, declaration).

Tree transformation can be viewed as the replacement of a tree, or a subtree, by another one according to the transformation rule (i.e. using pattern matching and substitution). The description of the replacement process via tree transformation is often easier, shorter, and intuitivly clearer than even a description in natural language, and permits the expression of explicit structures (i.e. does not require parsing). Yet for a wide class of transformation rules, this translation can be automated, thus providing a means for compiling abstract structures into functional programs [Ref. 1].

Abstract trees, representing both rules and programs, are constructed (i.e. parsed) and printed (i.e. unparsed), using what we call templates, which define the syntactic and the semantic structure of the concrete and abstract form of the rules and the programs.

Such a programming system does not require the user to have any kind of previous knowledge of a language syntax to write programs. Instead, he will be able to define his own language syntax by means of the templates. For example, a french speaking user might write the template "si -- donc -- autrement --" to describe an if statement, while a mathematician would prefer to write it as:

"-- if --, -- otherwise".

The templates, together with the transformation rules, make it possible to write programs in one form and have them printed in another form. For example a user can enter arithmetic expression using infix notation because it is easier and more natural, and the system will output the expression in postfix form for evaluation or other uses. Yet, although it is not our main objective, we can use this programming system to translate a program written in a special syntax to a legal form of a given language, or to convert one program structure to another one (e.g Pascal case statement into its equivalent sequence of if statements, or a while structure into a repeat structure etc.). In fact, we can imagine many other applications. However, in the last chapter we will discuss in more details the possible uses and advantages of this programming system.

B. APPROACH

Our approach for the development process of this programming system and its environment will consist in, a first step, to define the different objects on which the user and the system will operate. In the next step, we will define and describe the different phases for developing tree transformation programs and suggest what we think might be the appropriate tools which will assist the user at each phase. In the next, step we will investigate each one of

these suggested tools to determine the kind of functions and facilities they should provide including an analysis of the design tradeoffs. We will examine the interaction between the modules of the system and to what extent they must be integrated, the style of communication between the user and the system, and how flexiblity and friendliness can be achieved. We will also discuss some of the implementation aspects and describe briefly how the system is actually implemented. Finally, we will conclude the thesis with a discussion of the possible use of this system, its limitations, and we will suggest what we think might be useful extensions.

II. DESCRIPTION OF THE PROGRAMMING SYSTEM AND ITS ENVIRONMENT

A. GENERAL

In this chapter we will define the objects on which the user and the system will operate, then we will present a typical scenario of the different steps for developing programs, and finally we will describe the specifications of the programming environment and define a collection of tools which will be investigated and partially implemented.

B. DEFINITIONS

1. Template

A template, in our system, is a predefined formatted pattern of symbols. It comprises key words and place holders. Key words are used to improve the readability; place holders identify the location of variables to be filled when a copy of the template is used to construct the rules, or the programs. Thus, key words affect only the concrete forms of the rules and programs, while place holders affect both the concrete and the abstract forms. For example, two of the possible models of a conditional statement can be written as:

IF -- THEN -- ELSE -- or as: IF -- THEN --,-- OTHERWISE

IF, THEN, ELSE, OTHERWISE represent key words and the double dashs represent the place holders. Notice when the template is instantiated place holders can hold copies of other templates, thus, providing an unlimited level of templates nesting. Notice also that both forms of the "if" template are semantically identical, only the syntax is different.

a. role of the template

The role of the template is to define the concrete and the abstract models for the basic structures we will use to construct the rules and the programs. They will also provide a means for the user to construct syntactically correct rules and programs. That is, in some way, templates describe the grammar for the language used to write the rules and the programs. Each template is referred to by its name, which must be unique within the list of templates, so the user and the system can identify it without ambiguity. The following are templates which are going to be used as examples for the rest of this chapter.

template name	template text
tempif	if -- then -- else --
tempfact	factorial --
tempadd	-- + --
tempsub	-- - --
tempmult	-- * --
tempequ	-- = --
tempexpo	-- expo --

2. Transformation Rule

a. concrete transformation rule

Concrete transformation rules describe the permitted rearrangements of symbols and string substitution without using any semantic knowledge of these symbols. Each rule has two parts. The left part, called "analysis", describes the source string of symbols. The right part, called "synthesis", describes the target string of symbols.

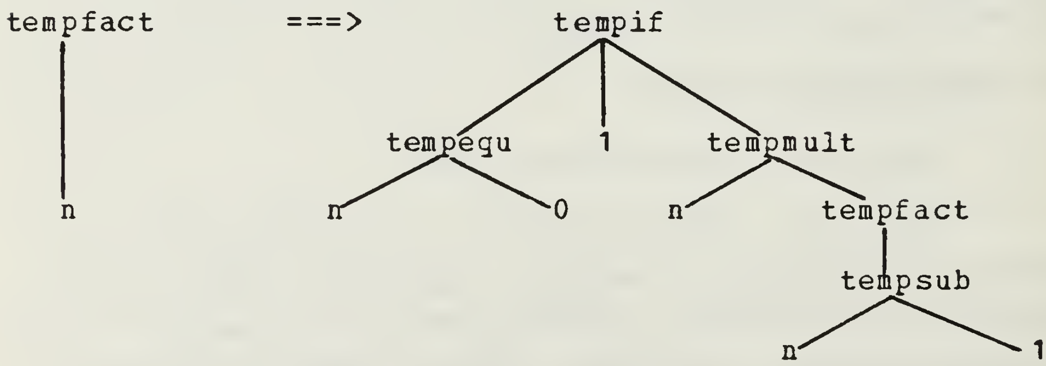
The two parts are separated by a double arrow to make the rule more readable. In our system, we will use the concrete form just to provide the user with a more convenient and natural representation of his abstract rules. The following are examples of concrete transformation rules:

example1: factorial n ==> if n=0 then 1 else n*fact n-1
example2: x expo m * x expo n ==> x expo m + n

b. abstract transformation rule

Abstract transformation rules are the same as concrete transformation rules except that they describe the permitted tree substitutions. Thus, the replacement process is done using the knowledge of the abstract structure of the tree via the templates used to construct these trees. Like concrete transformation rules, abstract transformation rules have two parts. The analysis part represents the source tree, and the synthesis part represents the target tree, that is, the one which will replace the program subtree when it matches the analysis part of the rule.

Using example1 given for the concrete rule, we now represent it abstractly as follows:



Notice that key words don't appear in the abstract tree because, as we will see later, they don't play any role in the matching and synthesization process since

they don't have any semantic meaning for the system. This, of course, has many advantages, such as to reduce the space to store the trees, reduce the time necessary for the matching process (i.e. less nodes to be matched), and eliminate a class of errors resulting from a difference in the spelling of these key words in the rules and the programs.

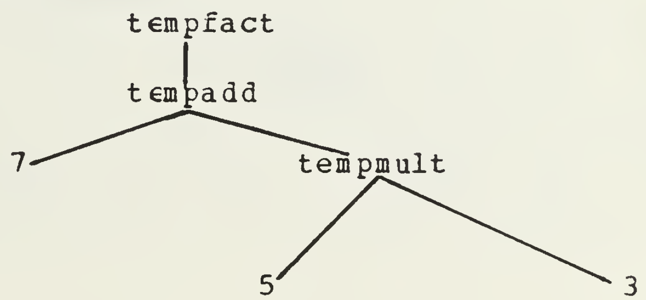
As we said earlier, both the concrete form and the abstract form of the rules are constructed via the templates. That is, the parsing and the unparsing of the rules is made more systematic. For example, from the if template the system knows that the corresponding tree will have a root referred to as "tempif" and three sons. Also, given the same tree, it will be able to construct its corresponding concrete form, by filling the place holders of "tempif" with the values of the sons from left to right.

3. Programs

Like transformation rules, programs have two forms, the concrete form and the abstract form. The concrete form is a pretty_printed text consisting of reserved words and constants (i.e literals and numbers). The abstract form is a tree whose main root and subtree's roots are names of templates, and whose leaves are constants. The two examples below illustrate a concrete form and its corresponding abstract form.

concrete form : factorial 7 + 5 * 3

abstract form :



4. Tree Matching and Variable Binding

Tree matching is done by comparing a program subtree with the abstract representation of the analysis part of the rule. As this process goes, the variables of the analysis will be bound to the values given in the program, which may be single constants or a whole subtrees.

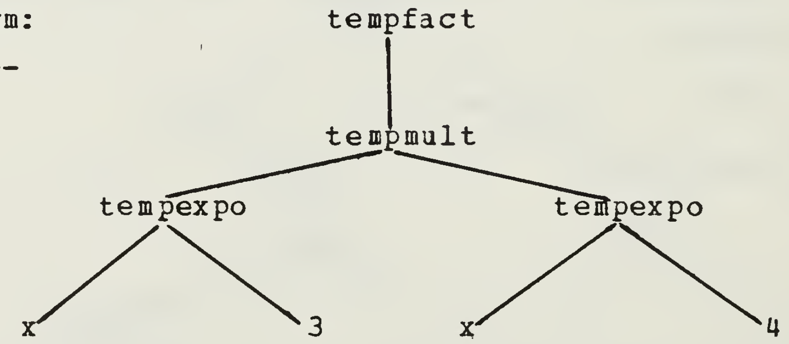
The variables and their bound values will form the context which will be used during the substitution of the program subtree by the synthesis part of the rule, but only if the matching had succeeded.

5. Synthesization and Tree Substitution

When a match occurs between the program subtree and the analysis part of a given rule, we will proceed to the replacement of this subtree by the synthesis part of the same rule. In this process the variables will be replaced by values to which they are bound in the context created during the matching process. We will refer to this operation as the synthesization. For example, given the following program:

concrete form: factorial x expo 3 * x expo 5

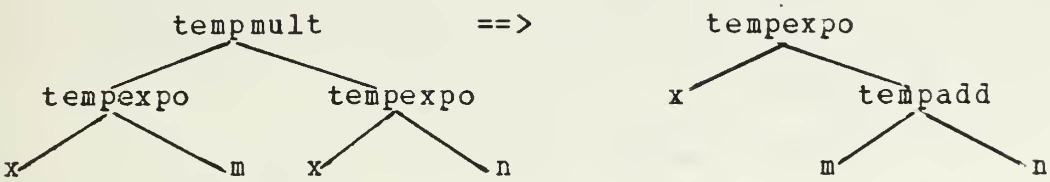
abstract form:



And given the following rule:

concrete form: x expo m * x expo n ==> x expo m + n

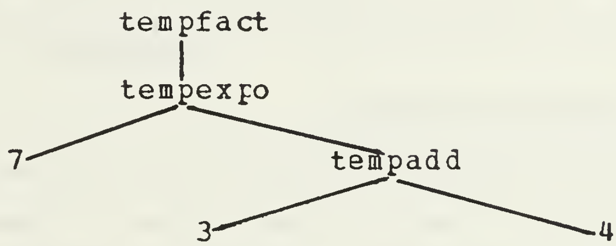
abstract form:



Since the analysis part of the rule matches the "TEMPMULT" subtree, the program will become after synthesization as follows:

concrete structure: factorial 7 expo 3 + 4

abstract structure:



With x, m, and n bound respectively to 7, 3, and 4.

Note that the concrete form does not tell us in what order we evaluate the different operations. Therefore, we need more specifications if we want to have a correct interpretation of the concrete rules. On the other hand, the order is explicitly represented in the abstract structure of the rule.

6. Built-in Templates and Rules

a. built_in templates

Built_in templates are an integral part of the programming environment. They are provided to define the structure of the built_in rules. Thus, when the user wants a built_in rule to be applied to a part of his program, he must use the appropriate built_in template to construct this part of the program.

b. built_in rules

Built_in rules are basic rules which can be applied to the program in the same way the user rules are applied. Like built_in templates, they are integrated in the system so they can be directly and efficiently applied to the user program.

C. DESCRIPTION OF TYPICAL PROGRAMMING SCENARIO

In this section we present a typical scenario describing the different steps to construct a program and its environmental context constituted by the templates and the rules.

1. Templates Creation

In this first step the user creates the set of templates necessary for his application. These templates will constitute the syntactic and the semantic framework to parse and unparse both the rules and the programs. Each template will have a unique name, and a body composed of a combination of key words and place holders (i.e. double dashes). The set of templates can constitute one or several lists.

2. Rule Constructions

Once the necessary templates are created, the user needs to create the transformation rules which describe the permitted tree substitutions. Each rule includes two parts, the analysis and the synthesis. Both parts are constructed using the templates created in the previous step. Thus, the user is not required to memorize the structure of the template. Instead, he issues a request for the template using its name, and the system will provide him with a copy of this template showing him (in inverse video) the place holders he must fill.

The place holders are indexed by a number followed by a letter. The number represents the level of nesting, which in fact corresponds to the height of the tree. The letter represents the position from left to right within the same level. The system keeps asking for more input until all the place holders are filled at all levels. When this requirement is satisfied the system will automatically signal the end of tree construction, and give the prompt for the next step.

3. Writing the Programs

Programs are constructed and written in the same way rules are. However place holders can now be filled by either another template or a data value (i.e. numeric constant, or litteral constant). In fact, we use this system to enter programs and rules in a way that is very similar to using a syntax directed editor to write conventional programs.

During one session the user can write many programs giving each one a different name. All the programs together constitute a list which, as we will see, can be processed as a whole.

4. Program Interpretation

Having completed the above steps, the user can now attempt the interpretation of his programs one at a time. When the interpretation is completed the system will automatically unparse and display the result. In addition, if the user desires, the result can be saved, or printed on a disk file.

D. REQUIREMENTS SPECIFICATION OF THE PROGRAMMING ENVIRONMENT, TOOLS DEFINITION

1. Specifications Description

Based on the typical scenario in the previous section, we now extend these ideas and investigate in more detail the environment which will support the user at each phase of the programs development process.

a. creating, and editing templates

. template manipulation

Earlier, we described a template as a combination of key words and place holders with a variable length. Therefore, when entering a template, we need a means to notify the system about the end of the template text. In addition, each template must have a name which will constitute its access key. Thus, we need a tool which supports these properties and provides a collection of utilities to create, access, and edit the templates.

Editing templates includes all conventional operations such as insertion, modification, deletion, displaying, and searching.

. List manipulation

Since templates will be grouped by lists, therefore we need facilities to manipulate the list of templates as a whole. That is, operations to save, restore, remove, and print the template lists. It is also useful if we can merge two lists of templates which, for some reason, have been created separately, or perhaps by different users. To perform such operations, the system should be able to handle alternatively several lists simultaneously present in the memory. This, of course, will increase the complexity of the system, since the system and the user have to keep track of which list and template are currently being edited. In addition, we need facilities to move from one list to another as well as for getting information about the current situation. We think, at this stage, it is to early to predict all the implications this might have on the design of the system and the command languages. Thus, this idea needs to be investigated more based on the analysis of the above tradeoffs.

b. creating and editing the rules

Editing rules includes operations such as insertion, deletion, modification, displaying, and printing. A set of rules constitutes a list which, can be manipulated by the user as a whole. Therefore, the system should provide facilities to create, save restore, display, and print the rules lists. Also, we may need to merge two lists, therefore we have to design the system so it can handle several lists of rules present simultaneously in the memory. However, this will depend on the results of the experimentation of the corresponding idea with the templates.

c. accessing the rule

Like a template, a rule is referred to by a unique name. However, in this case the name is not as important because it is used just to ease and shorten the search during the editing of the rules, to show the trace during the interpretation, and to avoid printing the entire rule.

The alternative solution would be to access a rule either by its position within the list, or sequentially, or by pattern matching. This solution seems to be more flexible, but it will result in a more complicated implementation without a lot of gain, because very often the list of rules will not be very long, and the user can give meaningful names to the rules such that it is easy to memorize them . Nevertheless, it is still possible for him to access the rules sequentially.

As described earlier, the body of each rule consist of the analysis part and the synthesis part. Every time the user wants to insert a new rule, the system will ask him to enter the name, then the analysis part, and finally the synthesis part.

d. constructing and parsing a rule

At this point, it seems opportune to emphasize the role the templates will play during this step. In fact they constitute the foundation of the programming system because they will guide the user in writing the rules, and the system in constructing the abstract trees, unparsing them, and displaying the concrete form of the rules. Each time the user request a copy of the needed template, while writing the rule, the system will assist him by showing the structure of the template, with the place holders highlighted (e.g. in inverse video), and indexed as described in

the previous section. At the same time the system prepares a copy of the corresponding abstract tree (which it already knows through the template), and starts filling the nodes with the values entered by the user at the right moment and places. Thus, it is unlikely that the user can omit providing the value of a place holder (i.e. node for the abstract tree). Also there is no way for the user to construct incorrect or incomplete structures because the system will not continue, and will keep asking for more values until satisfaction is obtained.

e. unparsing and printing the rule

This approach will also help the user to have a clear idea about what will be the structure and the shape of the abstract tree, things which may later facilitate the debugging of the rules. However, in practice, common users prefer to see a rather more readable and natural form of the rules. Thus, it is necessary that the system can unparse the abstract form and display a pretty_printed concrete form. This process is quite simple and straightforward, since all the system has to do is to get the appropriate template and just fill the place holders in the same order. However this raises some problems to be considered carefully. For instance, suppose that the user makes some modifications on the template affecting their structure, or completly delete some of them. Obviously,the system will not be able to unparse the tree, or may even give a completly unexpected concrete form.

The situation is similar to the mine expert who tries to take the mines out of a mined field with a modified plan. Of course we can imagine what will happen. Therefore, it is important that the user provide the system with the appropriate set of templates during the unparsing process, and if he modifies them on purpose (e.g. to change

the order of the place holders, or improve the readablity), he must be sure that this does not create an inconsistency between the abstract tree and the structure of the template. However the system must assist the user, and tell him whenever it detects such inconsistency, or give a special signal when the appropriate template is not found (e.g. precede the unparsed subtree by a special character).

f. Writing, and editing the programs

Programs are created and written in the same way as the rules are, except that a program includes only one part. Like the rules, programs have a concrete and an abstract form. The concrete form is what the user can display and see on the screen or printed on the paper. The abstract form is the corresponding tree constructed by the system at the same time the program is entered by the user.

The program is built up by putting together basic subtrees. The structure of these basic subtrees is given by the copy of the template, explicitly requested (i.e. using its name) by the user, and whose place holders are filled with constants, variable names, and other template names.

The advantage of such approach is to simplify the parsing process, since the abstract tree is constructed in parallel with no scanning and token recognition required. Also, it will guarantee that only correct programs are entered, because the system will show the template requested, and the user is required to fill all the place holders. Thus we can think of the templates as a grammar for the language and we are using a syntax directed editor.

Programs can be modified, displayed, saved and restored, pretty_printed on disk or on paper, and deleted. It is also useful that the user can move around the different parts of the program such as to go from a subtree

to another in the same level, zcoming in and out, searching for a given node, and altering the value of a node, especially during the debugging process.

g. interpreting programs

Programs are interpreted in their environmental context constituted by the templates and the rules. Thus a given program can be interpreted differently in different contexts. However the interpretation process itself is independent of the context. It consists in the successive applications of the user's and the built_in transformation rules.

The order of selection of the rules is sequential. However, conceptually the order in which they are selected and applied should be irrelevant to the final result of the transformation. A rule is applied to the program when its analysis part matches a subtree of the program. This will include: the matching itself, variable binding, synthesization and substitution.

The interpretation will end when no rule can be applied, then the result will be displayed. In addition, it should be possible for a user to request that the result will be sent to a disk file, so it can be printed or reused for further transformation with a different set of rules. During the interpretation the user might be interested in having the trace of the transformation, therefore the system must provide an option which allows visualizing the names of the applied rules and the intermediate results.

h. debugging programs

In order to assist the user in debugging his programs, we need facilities which make it possible to show the rules applied for the transformation, and to perform the interpretation step by step, or until a given rule is

applied, or a given situation arise, or a condition is met. The system should be able to answer specific questions, such as which rule is never used, or the number of times a given rule was applied, or why a given rule can not be applied to a given subtree of the program (i.e which nodes cause the failure of the matching process). The system must also detect some classic errors such as undefined node, nonnumeric argument for arithmetic operation, and other type checking. If an error occurs during the interpretation, the user is informed so he can stop the process, ask to show the result, or alter the value of a node and ask the system to continue.

i. communication language

Users communicate, via the screen and the keyboard, to the system by issuing commands using formatted natural language.

A command can be written on one line, or broken into a sequence of short subcommands. Thus the system will keep asking for additional information until it is able to execute the command or issue an error. This approach is based on the fact that a user who is familiar with the system will be able to handle long and complex commands, while a new user would prefer to be guided by the system, and see the same commands broken into a sequence of subcommands. No matter what the user types he should get a clear answer. Also, help must be proposed each time the user seems confused. Thus, at one extreme a user can memorize the entire command, and at the other extreme he needs just to know the name of the command or even just to request help. Help must be provided at the different stages, but only the relevant information should be given, because it is often difficult for the user to find the information he needs within several pages of displayed help.

2. Environment Tools

Our goal is to design an interactive programming environment which supports the programming system and meet the specifications described in the previous sections. Thus we will investigate, design, and partially implement the following tools:

1. A template editor
2. A rule editor
3. A program editor
4. An interpreter
5. A debugger

3. Integration

It appears from the specifications that the different modules of the system are closely related with a lot of interactions. Many resources will be commonly shared, such as templates (used by all the modules), rules (used by the rule editor and the interpreter), programs (used by the program editor and the interpreter). Many functions will be used by more than one module (e.g parsing and unparsing the abstract trees, searching for a giving rule or template, loading and unloading lists of templates rules and programs, and many other utilities).

In addition, we think it is important that users can switch back and forth between modules in an easy style without being forced each time to do the necessary file loading and unloading, and all the other routines to get the module started.

In my opinion we need an integrated system which provides enough security to prevent the user from making unrecoverable mistakes, and which gives clear and simple traffic indications so the novice will not lose his way through the modules. In summary, a secure, friendly, and flexible integrated environment.

4. Implementation Environment

Our system will be implemented on the PDP11/780 running a Unix system. We will use the Pascal language utilizing the Berkeley compiler. However, in order to maintain a certain portability, we have used only standard features.

III. SYSTEM DESIGN

Figure 3.1 represents the general architecture of the programming environment. The sclid arcs represent the data flow, while the broken arcs represent the control flow.

The environment is built up out of a number of modules and files. The modules are the template editor, the rule editor, the program editor, and the interpreter. The files are provided to hold the templates lists, the rules lists, the programs, and the results of the interpreted programs. Note that, except for the template lists, for each category of list we need two files: one for the abstract form used exclusively by the system during the load operation, and the other one for the pretty_printing of the concrete form which provide a readable ccpy for the user and can be printed using The UNIX command.

A. THE USER INTERFACE

1. Description

The user interface is the first layer of access to the system. Its function is to initialize the different pointers to the lists, load the built_in templates, provide a general help on how to use the system and navigate between the modules, give the prompt to the user so he can select the module he needs, and finally to make sure that every thing created or modified during the session has been either saved, or approved to be destrcyed before the user can exit the system.

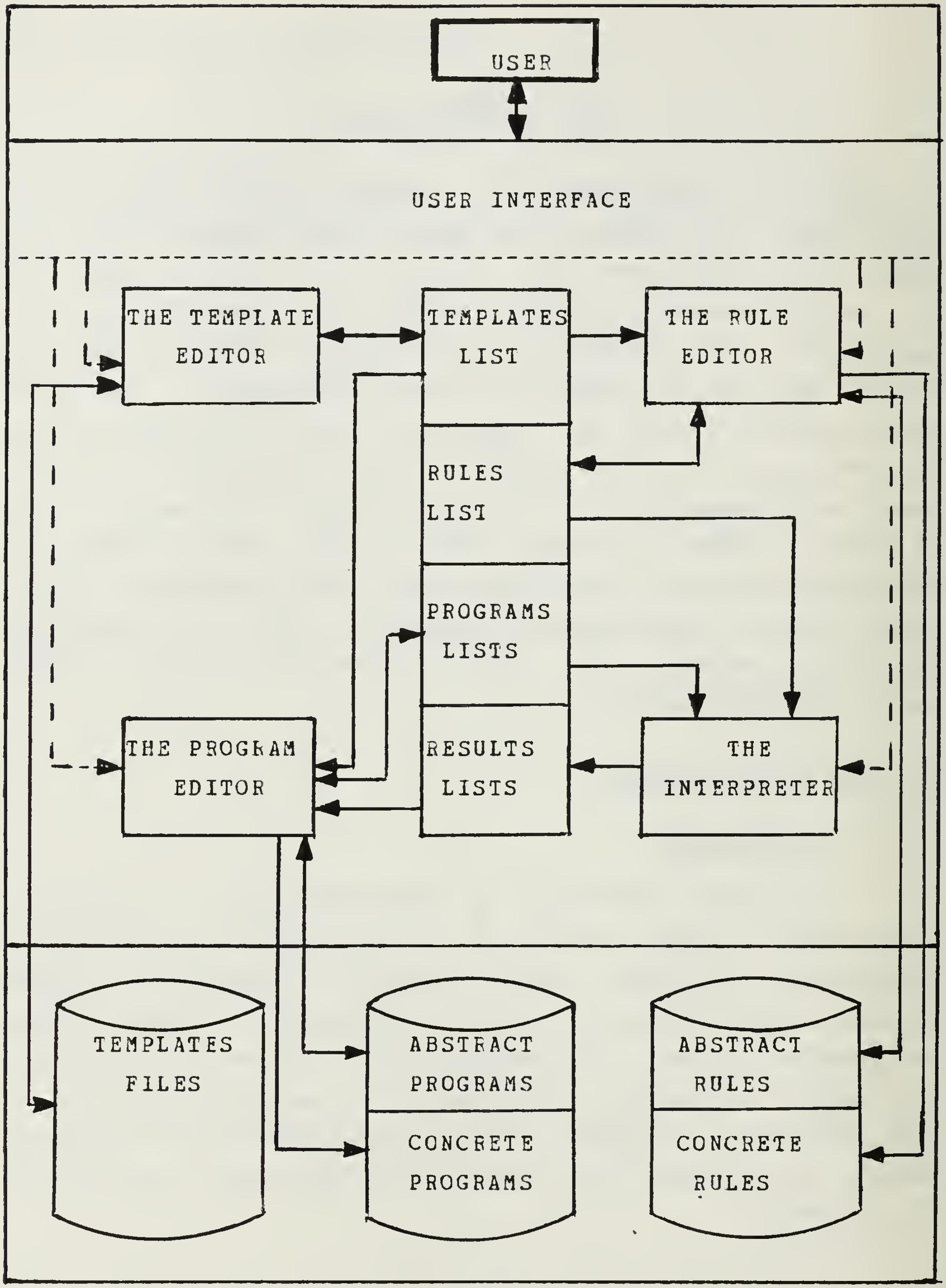


Figure 3.1 General Architecture of the System.

2. Module Selection

Once the prompt (-->) is given the user may select running either the template editor, the rule editor, the program editor, or the interpreter, by typing the name of the module. These are, respectively: TEMPEDIT, RULEDIT, PGMEDIT, and INTERPRET. After the selection is made, the user interface will give control to the selected module.

3. Getting Help

The user interface provides a general, simple help facility which gives user the essentials about the function of the system, how to select the appropriate module, and how to exit the system. This help may be obtained by typing the HELP command. However, for the user's convenience, the system will offer help at the beginning of the session or whenever an incorrect command is entered.

4. Ending the Session

The user can end the session and exit the system by typing the command QUIT. But, as a security measure, the system will check if there are lists of any kind created or modified during the session but not saved. If such lists are found, the system will display the names of these lists, and ask the user whether or not he wants to save them. A "NO" answer will confirm the quit and causes the system to terminate. On the other hand, if the answer was "YES" the system will re_issue the prompt signal, but it remains the responsibility of the user to save these lists by using the appropriate command offered by each module.

An alternative solution would be to let the system make the save automatically when the user answers "yes". But, there are at least two reasons why this solution is not preferable.

First, the user might not need to save all of these lists, but the system can not make such a decision unless explicitly told by the user. This will involve an extra interaction between the system and the user, which turns out to be the same as the first solution.

Second, and most important, the system will use the list name as file name, which can have a bad consequence, since any old file with one of these names will be automatically replaced by the new one, while the user may still need it.

In our design, as we will see later, the user will be given the choice to assign the name of the list to the saved file (i.e. overwrite the old file), or give a new name (i.e. create a new file). This allows him to have a family of files for a given family of applications.

We think the first solution is more flexible, more secure, and even in some cases more efficient.

5. Error Handling

An error occurs when the user types an incorrect command. That is, a command other than tempedit, ruledit, pgmedit, interpret, help, or exit. In that case, the system will echo the input command and print in inverse video the error message. It will also offer help to the user and re_issue the prompt signal for a restart. For example:

"temped: incorrect module selection"

"for more information please type help"

B. THE TEMPLATE EDITOR

1. General Description

Figure 3.2 represents the architecture of the template editor. The module is made of 11 functions which operate either on the list as a whole entity or on individual templates.

As we can see, many lists can be present simultaneously in the memory. However, The system and the user will edit one list at time (i.e. the current list). The user can switch back and forth between the different lists in an easy style with no load or unload required. The possibility to work alternatively on several lists in the memory adds more power and flexibility to the system. Lists or part of lists can be developed in parallel, perhaps for different applications. Yet, we will be able to take two lists, created by different persons, partially change them, and merge them to form a unique list to be used for a new application. This facility can also be helpful during the debugging process where we might need to try different versions of the same templates to decide which ones give the best results.

As a consequence of this design decision we need to include in the system features to keep track of which list is currently edited, what are the lists currently present in the memory, and for each one what is the current template. We need also facilities to move from one list to another and to remove a list from the memory. This also will require more checking and security measures and will result in more complex commands since each time we request an operation on a list we have to specify explicitly or implicitly (i.e. by default) on which list we want to perform these operations.

Of course we could decide that all operations requested will be done automatically on the current list, but then we lose the flexibility advantage because the user

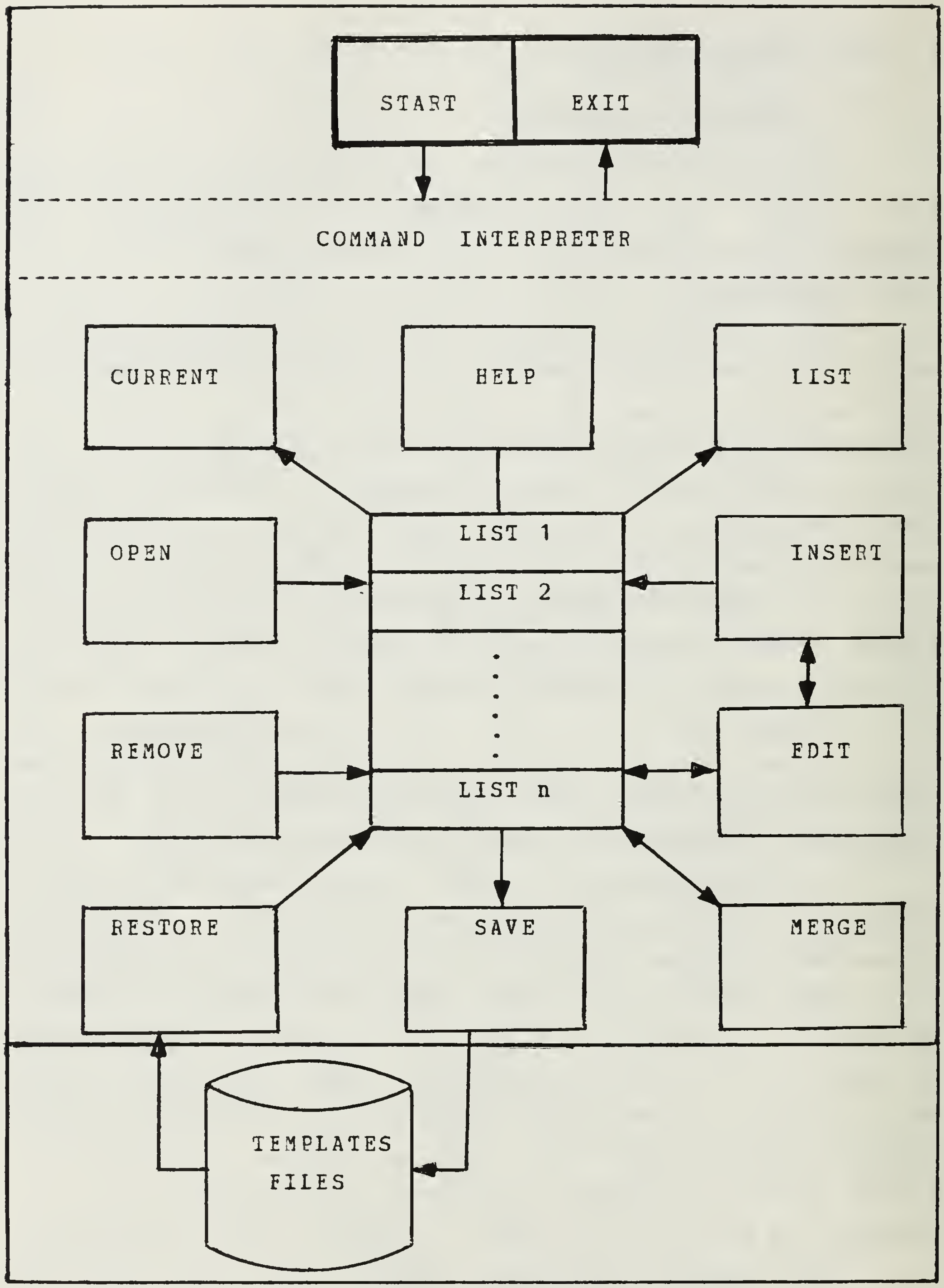


Figure 3.2 Architecture of the Template Editor.

will be required to use an open command in order to set the list he wants as the current one. This turns out to be in the user's view equivalent to a load, which we want to avoid.

In our system we will experiment with both approachs. That means we will implement the template editor such that we can have several lists in the memory, and implement the rule editor with the possibility of having only one list at a given time.

To keep track of which list and template is being edited, the system will maintain two pointers. The first one points to the current list, the second points to the current template within the list. The current list will be the last one processed by any operation requested by the user, except for remove, where the current list will become the next list, or "nil" if the removed list was the last. Likewise the current template will be the last one inserted, or displayed, or the one which follows the last one deleted, or "nil" if the one deleted happened to be the last template in the list.

The initial value of these pointers is "nil". However the system should be able to distinguish between an end of list "nil" and the initial "nil" (i.e. empty list), and make it clear to the user by giving an appropriate message such as "current template is nil: list is empty" or "current template is nil: end of list reached". At any moment the user may inquire about the current list and template, or about what are the lists currently in the memory with the number of templates each one contains.

The lists which are created or modified during the session must be protected against accidental deletion. This protection is removed as soon as the list have been saved.

2. Starting the Template Editor

The template editor is given control by the user interface when the command TEMPEDIT is selected. The module will display the prompt signal (TE-->) to inform the user that he can start entering the commands.

3. Command Interpreter

The role of this function is to accept the user's command, identify it and immediatly transfer the control to the appropriate function for analysis and execution. However, if the command is nct recognized the following error message will be sent to the screen:

"<Command>: is not a template editor command
please restart, or type help for more information"

4. Help

The user may ask for help, at any time the prompt signal is displayed, by typing the command HELP. However, like it was explained before, only the part concerning the template editor will be displayed, so it will be easier for the user to locate the information he needs.

5. Operations on Template Lists

In this section we will describe the different facilities provided to manipulate lists of templates as a whole entity. Also, we will describe the format of the different commands we will use to execute the different functions.

a. opening and locating a list

This function will allow the user to initialize a new list or locate an existing list and make it the current one. Basically, we will use the same command to

create a new list or locate an existing one. It is up to the system to determine which, depending on whether the list already exists or not.

Opening a new list will involve the following actions:

- Create a new list entity
- Assign the given name to the list
- Set the template counter to 0 (i.e. list is empty)
- Set current list = the new list
- Set current template to "nil"

Locate an existing list will involve the following actions:

- Set current list = the given list
- Set current template = first template of the list
  or "nil" if the list is empty

List names can contain from 1 up to 10 printable characters, which seems to be a reasonable size, and can be changed easly if it happens to be insufficient. Notice that it is not possible to have two lists in the memory with the same name.

The function will be executed by the command OPEN followed by the name of the list. Once the command is executed the system will issue one of the following message:

1. <listname>: new list created (i.e. list did not exist)
2. <listname>: contains x templates (i.e. list found)
3. <Listname>:illegal name (i.e. contains nonprintable char)

b. editing a list

The role of this function is to start the editing session on the given list. Editing will include operations such as: insertion, deletion, search, modification, and displaying. All these operations will be performed on templates, therefore we will describe and discuss them in more details in the next section. The editing mode is started by the command EDIT followed by the

name of the list and the starting point for the editing. The ccmmand can be typed on one line or broken into a sequence of subcommands controlled by the system, which will assist the user by asking for the remaining information needed to execute the command. The name of the list can be omitted in which case the system will take the current list as default value for the name of the list.

The starting point of the editing can be the first template of the list, the last template, a user specified template, or omitted, in which case the default value will be the current template of the list. Table I summarizes the different formats of the edit command. The inputs are shown in capital letters while the system responses are written using small letters. We will use this same notation to represent the dialogue between the user and the system.

TABLE I

Format of the Command Edit

```
*============================================================*
*                                                            *
* format # 1                                                 *
* ------                                                     *
*      EDIT <LISTNAME> FIRST | IAST | TEMPLATE | CR          *
*                                                            *
*                    ... edit mcde ...                       *
*                                                            *
* format # 2                                                 *
* ------                                                     *
*      EDIT <LISTNAME>                                       *
*                                                            *
*      enter starting point: FIRST | LAST | TEMPLATE | CR    *
*                                                            *
*                    ... edit mcde ...                       *
*                                                            *
* format # 3                                                 *
* ------                                                     *
*      EDIT                                                  *
*                                                            *
*      enter listname to edit: <LISTNAME> | CR               *
*                                                            *
*      enter starting point : FIRST | LAST | NAME | CR       *
*                                                            *
*                    ... edit mcde ...                       *
*                                                            *
*============================================================*
```

. Preliminary controls

Before starting the editing session the system will go through series of controls to check if the given list and template exist, if the list is not empty, and if the current template is not "nil" (i.e. end of the list reached). When no error occurs as a result of these controls, the system will acknowledge the request and print a header with the name of the list and the number of templates it contains, and then start the session by displaying the template which corresponds to the starting point. To exit the edit mode we need to press the escape key, which will return the control back to the command interpreter.

However, if the system detects an error during the preliminary controls, it will echo the command and print an error message as indicated in table II.

c. saving a list

In order for the user to have a permanent copy of his lists of templates which can be reused or printed, we need to have a function which allows him to save a list on a disk file in readable and pretty_printed form. Therefore, saving a list will be a quite similar process to the display except that we need to write all the templates of the list starting from the first one and finishing at the last one.

The command to execute the function will be simpler than the edit command, since all we have to specify is the name of the list and optionally the name of the file, if for some reason we want a different name (e.g. a file with the same name already exists and we don't want to overwrite it). The syntax of the command is: SAVE followed by the file name and the list name.

Both the name of the list and the name of the file may be omitted. In that case, the system will take the

TABLE II

Error Types and the Corresponding Messages for Edit

error type	error message
no list in the memory	no list has been created or restored
list does not exist	<listname>: list not found
empty list	<listname>: list is empty only insertion is allowed
template does not exist	<templatename>: template not found
current list is nil	current list is nil use the open command to set the current list

default values as respectively the current list and the name of the list. That means, if the file name is not specified the system will assign the name of the list to the new file. Table III shows the different formats of the command.

At the end of the execution the system will give a message to indicate that the function had been performed properly. In addition, it will print the name of the list saved, the number of templates it contains, and the name of the newly created file. Example:

"list1 containing 12 templates saved as file1"

If, on the other hand, an error was detected the command will not be executed, and the user is notified with the appropriate error message as indicated in table IV.

TABLE III

Format of the Command Save

```
format # 1
------
  SAVE <FILENAME> <LISTNAME>

format # 2
------
  SAVE <FILENAME>

  enter the name of the list: <LISTNAME> | CR

format # 2

------
  SAVE

  enter the name of the file: <FILENAME> | CR

  enter the name of the list: <LISTNAME> | CR
```

TABLE IV

Error Types and the Corresponding Message for Save

error type	error message
no list in memory	can not save:nc list in the memory
list does not exist	<listname> : is not found
current list is nil	can not use current list: value is "nil" please use open to set current list

. structure of the template file

A template file must have a structure which allows the system to reconstruct easly and systematically

the list and its templates in the memory during the loading. This structure will be a simple mapping of the memory list onto the disk file. Thus, template files will have a name which includes a maximum of 10 characters, and will contain the templates in the order of input.

Each template is composed of 10 characters name (if necessary completed by trailing blanks), followed by the body written like a conventional text, but terminated by the character escape. Figure 3.3 shows a sample print of a template file.

```
tempeval   : eval --
tempexpo   : -- expo --
tempif     : if -- then -- else --
tempfact   : factorial --
tempmult   : -- * --
tempadd    : -- + --
tempsub    : -- - --
tempdiv    : -- / --
tempequal  : -- = --
templess   : -- < --
tempgreat  : -- > --
tempevcon  : evalcon -- -- --
```

Figure 3.3 Printing of a Template File.

Notice that we have inserted a blank, a colon, and another blank between the name and the text to make the separation clear and the file more readable.

d. restoring a list

This utility allows the user to load in the memory files containing templates. The command which causes the execution of this function is: RESTORE followed by the name of the file (filename), followed by the name to be assigned to the list in the memory (listname). The name of the list may be omitted in which case the name of the file will be assigned to the loaded list. However, if the name has been already assigned to an existing list, the system will give a warning message and wait for a response from the user who may either order to overwrite the old list, or give another name, or quit for further investigation.

Table V summarizes the different formats of the command, and the resulting interaction between the user and the system.

. files created using other editors may be loaded

Files created using editors other than the template editor (e.g. ex) may also be loaded by the restore command. But, in that case, it is the responsibility of the user to make sure that the files have the appropriate structure the system expects to find. Therefore, files which are not formatted the way we have described in the previous subsection will cause an error during the loading, and the abandonment of the operation.

Table VI shows the different error types which may occur during the restore operation with the corresponding message which will be sent to the terminal.

As we notice, we did not include in the table the case where the specified file is not found on the disk. The reason is that we are faced with many alternatives to deal with this problem. In the rest of this subsection we will present these solutions, and discuss the advantages and disadvantages of each one.

TABLE V

Format of the Command Restore and System Responses

```
format # 1

  RESTORE <FILENAME> <LISTNAME>

format # 2

  RESTORE <FILENAME>

  input listname or cr if using same name:<LISTNAME> | CR

format # 3

  RESTORE

  input filename: <FILENAME>

  input listname or cr if using same name:<LISTNAME> | CR

                    ---------------------

  {the following dialogue will take place if the lists
   already exists }
   WARNING ! <listname> already exists in the memory
   do you want to overwrite it yes/no : YES | NO
   { "yes" the system will overwrite the old list }
   { "no" the system will continue the dialogue }
   do you wish to give another name yes/no: YES | NO
   { if yes the system will ask for the new name }
   input listname:<LISTNAME>
   { if no the system will abandon the request }
```

. solution 1

The first obvious solution to this case would be to issue an error message indicating that the file does not exist on the disk, and then give the control back to the user, so he can either restart the command with the right

TABLE VI

Error Types and the Corresponding Messages for Restore

error type	error message
missing space after t.name	missing space after the template name
missing ':'	missing colon after the template name
missing space before t.text	missing space before template text
eof reached but no escape	unexpected end of file

file name, or save whatever he has in the memory and exit for more checking, or simply abandon the operation and start a different task. Unfortunatly, this solution presents some difficulties due to the implementation environment.

First, in the Berkeley Pascal environment, it is not possible for a user's program to get back the control when a "reset" fails. Instead he will get an error message, but the control is automatically returned to Unix without having the chance to save the content of the memory.

Second, Pascal, like most languages, does not provide an instruction which allows one to check if a file exists before we attempt to open it.

Therefore, in order for the solution to work, we must write a special routine. This routine will read the directory and return a flag which can be tested to find out whether or not the given file exists, and subsequently either attempt the open, or issue an error message but return the control back to the program.

. solution 2

The second solution would be to let the system save automatically the lists created or modified during the session before it attempts to open the file. Thus, if the user loses the control because of an open failure, he will be able to use these file copies for backup.

Although this solution is feasible, it will affect considerably the efficiency of the function and may perhaps encourage the user to be careless when he enters the command. Therefore it is preferable to make the save rather optional. That is, when the user requests a restore operation the system will send a warning message to remind him about the possibility of losing the control, and ask him if he wants to save the lists created during the session. The user can either accept the offer and save the lists he wants to, or change the file name if it was incorrect, or confirm the command in which case the system will go ahead and attempt to open the file.

. solution 3

The third solution requires the user to declare at the beginning of each new session the files which are likely to be loaded. Thus, there will be no risk resulting from losing the control since there is nothing in the memory yet.

The problem is that the user has to know in advance the input files he will use, which is not always the case. Also, he will not be able to restore files created and saved during the session unless he exits the system and starts all over again, which is not practical, and may take a rather long time to do.

. conclusion

It seems clear that the first solution offers the maximum flexibility and security. There is no possibility for the user to lose the control or the content of the memory because of a misspelling in the file name. But, in turn it requires more work to be implemented, and extra time for the execution of the function, since the system has to check the directory, even if the user had input the correct file name. Also, this solution will not be general because the routine will not work in a different environment with a different directory organization.

The second solution requires less work but it does not provide the maximum security since, from experience, we know that users get very quickly tired of warning messages and stop giving them any attention which, in this case, may cause the loss of important information and hours of work.

The third solution offers enough security even though we can still lose the control, but since this will happen only at the beginning of the session there will be no loss of memory content. But, on the other hand, it will put some inconvenient restrictions, and could be a time consumer.

In our system we will use the second solution because it represents a reasonable compromise between the efficiency, the amount of work for the implementation, the flexibility, and the security. Also, it is a general solution which will work if the system is run in a different environment.

Table VII illustrates the interaction between the system and the user for the second solution.

TABLE VII

User System Dialogue before a File is Opened

```
*==============================================================*
*                                                              *
* WARNING ! if no such file exists the system will abort       *
*                                                              *
* and the content of the following list(s) will be lost        *
*                                                              *
*          { listing of the lists not saved }                  *
*                                                              *
* do you wish to save them yes/no : YES | NO                   *
*                                                              *
* { a "yes" answer will cause the abandon of the restore       *
*                                                              *
*   operation so the user can use the command SAVE to          *
*                                                              *
*   save the lists he wants and restart the  operation }       *
*                                                              *
* { a "no" answer will confirm the request and let the         *
*                                                              *
*   system attempt to open the file and hopefully respond      *
*                                                              *
*   <listname> restored from file : <filename>                 *
*                                                              *
*==============================================================*
```

e. removing a list

This utility will allow removing a list from the memory, and freeing the occupied space. This function may not appear very necessary since lists which are no longer needed may simply be ignored and not saved at the end of the session. But, in an integrated system like in our case, needs for memory space grow so rapidly such that a memory clean up becomes necessary.

Thus, without this utility, the only way to remove lists which are no longer needed and to make the space occupied available, is tc save whatever we will still need, exit the system, and start all over again with a clean and reorganized memory space. This, of course, will cost much more time than executing a simple command.

The command to execute this function is: <u>REMOVE</u>, followed optionally by the name of the list to be removed. However, The system will remove the current list when no name is specified.

Nevertheless, to prevent the user from accidentally removing a list created or modified during the current session but not saved, the system will check the update flag of the given list, and in consequence it may issue a warning message and give the control back to the user so he can abandon the operation or confirm it.

Table VIII shows the different formats of the command and the resulting dialogue between the user and the system.

TABLE VIII

Formats of the Command Remove and System Responses

```
*==============================================================*
*                                                              *
* format # 1                                                   *
* ------                                                       *
*                                                              *
*   REMOVE <LISTNAME>                                          *
*                                                              *
* format # 2                                                   *
* ------                                                       *
*                                                              *
*   REMOVE                                                     *
*   input listname to remove: <LISTNAME> | CR                  *
*                                                              *
*                         ---------------------                *
*                                                              *
*   { the following dialogue will take place if                *
*                                                              *
*     the given list to remove has not been saved }            *
*                                                              *
*   WARNING! <listname> has not been saved                     *
*                                                              *
*            do you want to continue yes/no : YES | NO         *
*                                                              *
*   { a "no" will cause the abandon of the operation }         *
*                                                              *
*   { a "yes" will let the system remove the list    }         *
*                                                              *
*==============================================================*
```

. error checking

Before starting the execution the system will go through series of controls to determine if the command can be executed properly. The types of error which might result

from these controls with their corresponding messages are given in Table IX.

TABLE IX

Error Types and the Corresponding Message for Remove

error type	error message
no list in the memory	can not remove:no list in the memory
list does not exist	<listname>: list not found
current is nil	can not use current: value is "nil" use open to set the current list

f. merging lists

This function takes two lists and forms a unique one by simply concatenating the second list with the first one. Thus, it is possible tc have duplicate templates in the resulting list which of course should not be allowed. However, instead of stopping the merge operation as soon as a duplicate is found, the system will notify the user by displaying the name of the duplicate template and then continue until the merging is ccmpleted. It is, therefore, the responsibility of the user to take the appropriate action to eliminate these duplicate.

The command to execute the merge function is: MERGE. Following the name of the command, the user may give

the names of the first list, second list, and the resulting list. Like the other commands, these names may be omitted in which case the system will take by default the current list for every nonspecified list. However, in any case the first list and the second list must already exist, while the third list can be either a new list, or an old list to be replaced. The latter case is treated as an overwrite. Thus, the system will go through the same kind of interaction with the user to get, if necessary, the confirmation for the overwrite.

The different formats of the command with the system responses are shown in Table X, while in Table XI we summarize the different types of error which may occur during the controls preceding the execution of the command.

g. listing the template lists

This utility function will give the user the listing of the template list currently present in the memory with the number of templates each one contains. However if there is no list it will display the message:

"no list in the memory"

the command for this utility is : LIST

h. inquiring about the current position

very often, after hours of work with the system, the user may get confused about which list is the current one. Thus, the system provides a utility function which gives him such informations, and even tells him about the current position within a list. The command to request this information is: CURRENT. The system will respond by one of the messages given in Table XII.

TABLE X

Formats of the Command Merge

```
*============================================================*
*                                                            *
* format # 1                                                 *
* ------                                                     *
*                                                            *
*   MERGE <LISTNAME1>+<LISTNAME2>=<LISTNAME3>                *
*                                                            *
* format # 2                                                 *
* ------                                                     *
*   MERGE <LISTNAME1>+<LISTNAME2>                            *
*                                                            *
*   input listname3: <LISTNAME3> | CR                        *
*                                                            *
* format # 3                                                 *
* ------                                                     *
*   MERGE <LISTNAME>                                         *
*                                                            *
*   input listname2: <LISTNAME2> | CR                        *
*                                                            *
*   input listname3: <LISTNAME3> |CR                         *
*                                                            *
* format # 4                                                 *
* ------                                                     *
*                                                            *
*   MERGE                                                    *
*                                                            *
*   input listname1: <LISTNAME1>                             *
*                                                            *
*   input listname2: <LISTNAME2>                             *
*                                                            *
*   input listname3: <LISTNAME3>                             *
*                                                            *
*                              ...........                   *
*                                                            *
*        { listing  of duplicate templates found }           *
*                                  .                         *
*                                  .                         *
*                                  .                         *
*   <listname3>,containing x templates created               *
*                                                            *
*============================================================*
```

6. Operations on Templates

In the previous section we described and discussed each of the operations which can be performed on the template lists as a whole entity. In the present section we will do the same thing for the template as an individual object. That is, we will describe the facilities provided to manipulate separately a single template.

First of all, we need to mention that all these operations will be executed on the templates of the list

TABLE XI

Error Types and the Corresponding Message for Merge

error type	error message
no list in the memory	can not merge:no list in the memory
list1 or list2 not in memory	<listname> not found
current list is nil	can not use current list is "nil" use "open" to set the current list

TABLE XII

Formats of the Information Message for Current

<u>message type 1</u>
<template name> in <list name>
<u>message type 2</u>
"nil" <list name> is empty
<u>message type 3</u>
"nil" end of list <listname>

currently edited, except for the insert operation which may be requested independently on the current list. Thus, before starting the insertion, we may need to set the current list as the one we will be inserting in. This can

be done explicitly using the open command or implicitly by means of the commands RESTORE, SAVE, EDIT, or MERGE since these commands have the side effect of opening a list and setting it as the current one.

a. displaying a template

As we said earlier, the editing session will automatically start by the display of the template corresponding to the starting point. Each template is displayed name first followed by the body, which may include any number of characters. Thus the text can be printed on several lines formatted in the same way they have been when entered by the user.

The user may either continue to display the templates sequentially by pressing the return key, or alter it by giving a new starting point in the same way we have described for the options in the command EDIT (i.e. first, last, or a template name). Also, these same options are available when the end of list is reached.

b. insert and append

Many editors treat insertion and appending as separate cases. In our system we provide a unique utility because appending is no more than a special case of insertion which happens either at the beginning or the end of the list. Thus, what will make the difference is either the option provided explicitly with the command, or the current position during the edit mode.

. direct insertion

The fact that the template editor knows which is the current list and where is the current position within the current list even when not in editing mode, with the fact that we are able to reference a template by its name,

makes it possible to do a direct insertion without being in the editing mode. However in such case the user must specify the place where to start the insertion (i.e. at the beginning, the end, before a given template, or before the current template).

This view holds that very often, the user knows exactly where he wants to insert, therefore there is nc need for him to waste the time searching the place of insertion (i.e to set the current template) by using the display facility.

. insertion from the editing mode

It also possible tc insert new templates while in editing mode. In that case we are not required to specify the place of insertion since it will be automatically assumed before the template currently displayed, or at the end of the list if it was the current position. Thus, the user can switch back and fcrth from the display mode to the insertion mode.

The command for insertion is: <u>INSERT</u> followed optionally by the place of the insertion. The place of insertion can take the same values than the starting place in the edit command (i.e. first, last, a template name, or omitted when before the current template).

Note that when the list is empty, the system will automatically start inserting at the beginning of the list except when a template name was given as reference for the place of insertion. In that case an error will occur and a message will be printed to indicate that the template was not found. Also, when nc place is specified and the current template is "nil" the system will start inserting at the end of the list. In any case before the user can enter the new templates the system will notify him by a message where the insertion will be. Thus, the user can say "OK

that is what I wanted", or if that was not what he expected the insertion to be, he can simply abandon the request by pressing the escape key.

. entering a new template

Entering a new template will require the user to input first the name, than the text of the template. The name must be unique, and may contain up to 10 characters ended by the return key. An error will occur if the name entered has been already assigned to an existing template, or when the name begins with a digit or a double quote (later will explain the reascn of these restrictions). Also, when the user inputs more than 10 characters for the name, the system will simply truncate to the tenth position. On the other hand, the text of the template may include any number of characters terminated by the key escape.

. ending the insertion

The end of the insertion will be notified by pressing the escape key in the place of the name. Thus, the last two characters of the insertion should be filled by the escape key. The number of templates inserted will be automatically displayed, and depending on whether the insertion was requested from the edit mode, or independently, the template editor will switch back to the edit mode, or to the command interpreter.

c. deleting a template

Template deletion will be done only during the editing mode using the subcommand DELETE. Thus, the user must display the template before he can issue a delete operation. This will provide more security since the user will see the template and have a final checking before he can delete it.

Once the operation is executed, the system will automatically display the next template, which becomes the current one. However, if the deleted template was the last one in the list, the system will signal the end of file and set the current template to "nil". In that case, we may either restart the display, or simply exit, as it has been explained in subsection 4d.

d. searching for a template

During the edit mode, it is possible to search for a given template by simply typing its name. When found, the template will be displayed and becomes the current one. However, if not found, the system will print an error message and return back to the previous situation.

e. 'a typical editing session'

Figure 3.4 shows a typical editing session where we have two templates inserted (tempfact, tempexpo), one deleted (tempadd), and five displayed (tempif, tempeval, tempadd, tempsub, and tempdiv). Also, the figure shows a successful search for a template (templeval), and another search for a template (tempi) which failed.

f. modifying a template

In our system, the only way to modify a template is to delete it and insert in its place the new one. This decision is based on the tradeoff between the frequency of modification, the average length of the template, and the complexity resulting from including a facility to modify partially a template with the effect that will have on the implementation. Nevertheless, we must admit that it is very unpractical and unconvenient to be forced to perform a delete and an insert just because we want to make a small

```
EDIT TEMPLIST1 FIRST
*=====================================*
*                                     *
*  templist1 contains 5 templates     *
*                                     *
*=====================================*
tempif     : if -- then -- else --
tempeval   : eval --
tempadd    : -- + --
DELETE
....tempadd deleted
tempsub    : -- - --
INSERT
             ..... insertion mode .....
.... insertion before tempsub
input name: TEMPFACT
input text: FACTORIAL -- "ESCAPE"
input name: TEMPEXPC
input text: -- EXPO -- "ESCAPE"
input name: "ESCAPE"
.... 2 templates inserted
.... back to display mode....
tempsub    : -- - --
tempdiv    : -- / --
"end of list reached
restart or escape tc exit": TEMPEVAL
tempeval   : eval --
TEMPI
....tempi not found
"ESCAPE"
....edit terminated
templist1 now contains 6 templates
TE-->
```

Figure 3.4   A Typical Editing Session.

modification on a given template. Therefore, we suggest the incorporation of a more elaborate and flexible facilities to modify, including things such as: pattern substitution, string insertion, and concatenation.

7. Exiting the Template Editor

The template editor will terminate by the command EXIT. The control is then returned back to the user interface without any further investigation. That means the user is not required to save the lists when he exists the template editor, even if those lists have been created or modified during the session. The reason is that it is still possible for the user to return back to the template editor for more work on the templates. In fact this situation of switching back and forth between modules will be very frequent during the debugging and the development of the application. Thus, it is better to delay the control until we are sure that we have the last version of the templates (i.e. when the user asks to exit the entire system). However, it would be better practice to save the lists before we switch to another module, just in case (e.g. we lose the control because of an infinite loop in the interpretation of the program).

8. 'Built_in Templates'

As we said in chapter 2, the system provides a set of built_in templates which are loaded at the starting of the session, and are grouped in a list called "BUILT_TEMP". These templates are listed in Figure 3.5

The first 8 templates defines the structure of the analysis parts of the built_in rules. Thus, if the user wants a built_in rule to be applied to his program he must use the appropriate built_in template either to construct

```
+          : -- + --
*          : -- * --
/          : -- / --
-          : -- - --
=          : -- = --
>          : -- > --
<          : -- < --
eval       : eval --
error1     : error: left argument is not numeric -- --
error2     : error: right argument is not numeric -- --
error3     : error: arguments are not numeric -- --
error4     : error: operands are not compatible -- --
```

Figure 3.5 Listing of the Built_in Templates.

the concerned part of his program so the built_in rule can be matched against this part, or to built the synthesis part of one of his rules so that the program can be transformed such that it contains a part which matches the built_in rule.

The rest of the templates are used to unparse the errors subtrees which may occur during the application of the built_in rules. That is when an operand of an arithmetic expression is not a numeric constant, or when the operand of a boolean expression are not compatible. In that case the system will replace the subtree by an error subtree whose root contains the name of the error template and whose two children are the operands. For example the boolean expression ' "A > 3 ' will be transformed to ' error4 "A 3 ' and will be unparsed as:

' incompatible operands for boolean expression : a 3 '

C. CONCLUSIONS DRAWN FROM THE DESIGN OF THE TEMPLATE EDITOR

Before we start the design description of the rule editor, which will be our next step, it is worthwhile to mention some observations and lessons learned during the design of the template editor. Although conceptually the two modules are quiet different, there are some common aspects which relate to the list manipulation and the user friendliness.

In the template editor we allowed many lists to be present simultaneously in the memory, and subsequently we included facilities and other security measures to deal with this situation. As a result we ended up with a relatively complex set of commands to manipulate lists. Each time the user has either to specify the list he will be working on, or let the system take the default value which may cause some surprises ("Oh I thought the current list was the one I inserted in!" But meanwhile he forgot about the save he requested on the other list).

Furthermore the number of checks and, therefore, the number of resulting errors have dramatically increased by a factor of 3, thus affecting the efficiency and the amount of interaction between the system and the user. Yet, in some situations we have been pushed to create a unsecure situations such as allowing duplicate templates in the same list (see MERGE), this may cause a lot of trouble during the construction of the rules or programs if the user fails to take the appropriate action to eliminate those duplicates.

Last but not least, this design decision will have a direct impact on the implementation, and of course, on the verification and debugging of the system.

In counter part we gained some power and flexibility in the system which are not yet proven to be useful. Our arguments to support that decision were:

1. The possibility for the user to develop several different lists of templates in parallel for a family of applications.

2. The possibility to try many different versions of templates for the same application in order to determine which ones give the best results in term of readability.

3. To be able to take two different lists, developed by different persons, or the same person for different applications, and make them a unique one for a new application.

Although these arguments are sufficient to motivate our decision, there is still an important unknown factor which depends on how much the user will take advantage of these facilities. Usually, common users tend to be rather conservative favoring simplicity even if it is not the most efficient way.

Based upon these observations, and the time required to implement such facilities we will design the rule editor such that the user is allowed to have only one list in the memory at one time. Thus, if the user wants to edit or try different lists of rules he must unload the old one and load or create the new one. However, the templates which go with these different rule lists may be present simultaneously in the memory.

D. THE RULE EDITOR

1. General Description and Module Architecture

Figure 3.6 represents the architecture of the rule editor. The module is made of 9 functions which operate on list, or on rules. However, since this module does not allow multiple lists to be simultaneously present in the memory, we don't need facilities to deal with such situation. Thus, the commands will be simpler than those of the template editor. Notice also, now there are two files: one for the abstract rules, and the other one for the concrete rules.

The plan of this section will be similar to the previous one, in the sense that we will first discuss and describe the design of the facilities which manipulate the list, then we will do the same thing for the the facilities which manipulate the rules. However, since a list of rules is treated like a list of templates we will try to maintain the same strategy and use the same commands. We will not spend much time describing again the same features and the necessary preliminary controls, instead we will focus on the specific requirements of rule manipulation.

2. Starting the Rule Editor

Like the template editor, the rule editor is given control by the user interface when the command RULEDIT is selected. The prompt signal will be displayed (RE-->) to notify the user that he can start interacting with the module.

3. The Command Interpreter

The role of this function is to accept the user's command, identify it, and when everything is correct, transfer the control to the appropriate function for execution. An error message will be sent when the command was not correct.

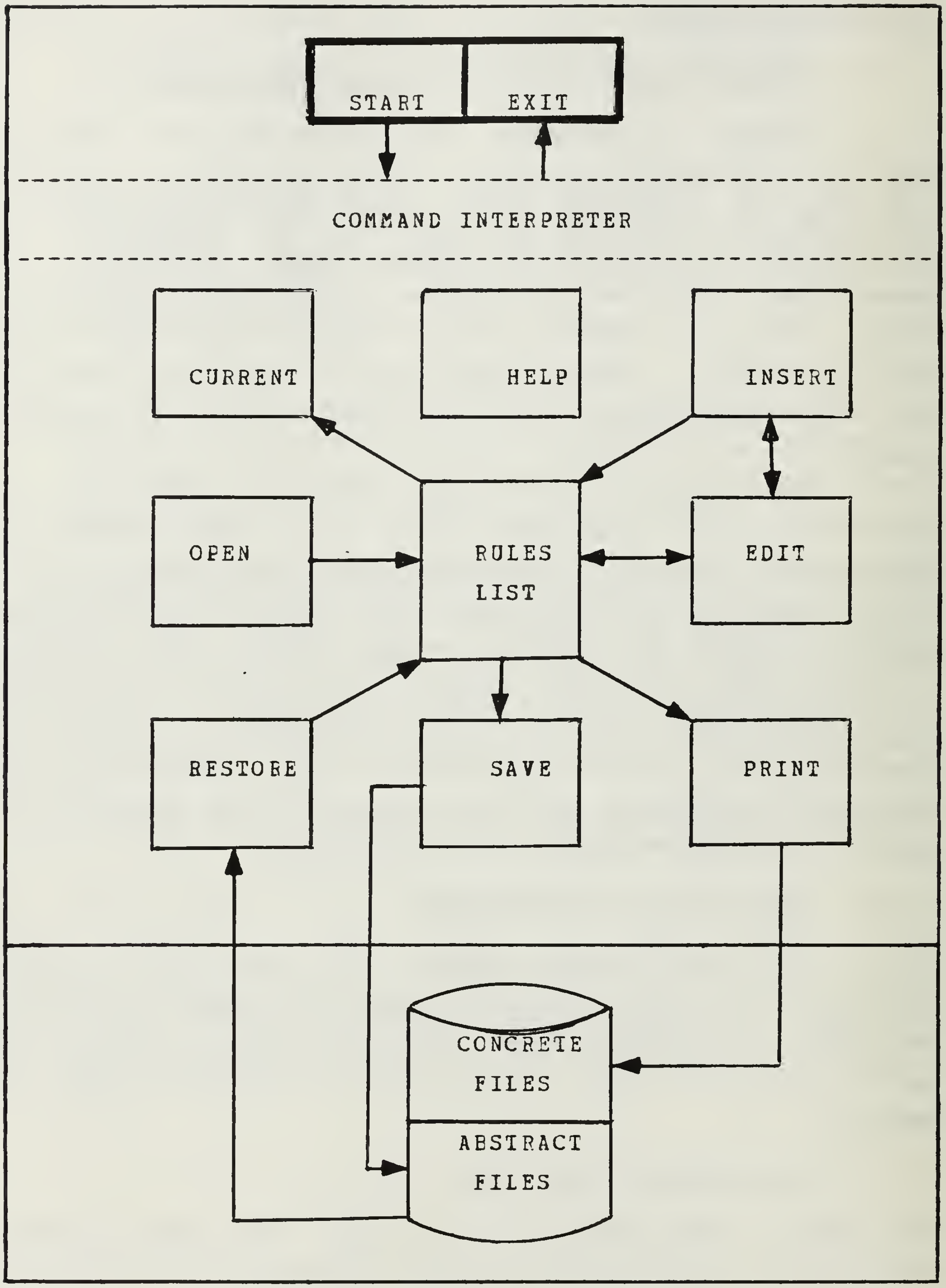


Figure 3.6 Architecture of the Rule Editor.

4. Lists Manipulation

In this subsection we will describe the facilities which allow us to process a list of rules as whole entity, the commands to request these facilities, and the different controls and checking needed before and during the execution of the function.

Obviously, facilities such as remove, merge, and list are not needed in the present case because at most we can have only one list. In addition, most of the commands will be shorter since now it is not necessary to specify the name of the list. On the other hand, because the rules have two different forms we need to include additional facilities to handle each one of these two forms. Also in some cases we need to include additional security measures.

a. the open function

Like in the template editor, the open function will create and initialize a new list. However, now the system has to deal with different situations, which are:

1. There is no list in the memory
2. There is a list but it is empty
3. There is a list which contains rules

The first two cases don't require any special treatment other than to create the list, initialize or reinitialize it, and assign to it the given name. The third case requires more consideration from the system. Before it can execute the OPEN the system must make sure that the real intention of the user is to reinitialize the list. Thus it must inform him about the existance of the old list, its contents, and whether or not it has been saved after the last modification. Based on this information the user can either abandon to the request or confirm it, in which case the system will go ahead and reintialize the existing list.

Table XIII shows an illustration of the interaction between the system and the user for the third case.

TABLE XIII

Dialogue for the Reinitialization of an Existing Rule List

```
*=======================================================*
*                                                       *
*  OPEN LISTNAME                                        *
*                                                       *
*  warning! There is already a rule list in memory      *
*                                                       *
*           <listname> not saved since the last change  *
*                                                       *
*           do you want to reinitialize it yes/no:      *
*                                                       *
*   1. A "yes" answer causes the reinitialization       *
*                                                       *
*   2. A "No" answer causes the abandonment of the      *
*                                                       *
*      request                                          *
*                                                       *
*=======================================================*
```

b. the edit function

Conceptually there is no difference between editing a list of templates and editing a list of rules. Therefore we will carry out the same operations we described before. These operations are display, delete, insert, and search. Nevertheless, we must mention that in the present case the user doesn't need to specify the name of the list to edit, since it will be automatically done on the current list provided it had been previously created by an open command, otherwise the system will generate the following error:

"can not edit, no rule list opened"

The major difference, though, will be in the way we will enter a new rule. But since this operation is part of rule manipulation, we will delay its discussion until the next subsection (i.e rule manipulation).

c. saving and printing a rule list

Although these are two separate functions, we prefer to discuss them together because they represent two symmetric aspects of the rules.

As we already know, a rule has two forms: an abstract form and a concrete form. The abstract form is a tree structure which will be used for the transformation process. The concrete form is what will be displayed by the system for the user convenience.

Obviously, like any other software product, it does not pay to spend a lot of work to develop the rules if we can not reuse them. Therefore it must be possible to have a permanent copy of the rules which can be restored when needed. The question is which form we will save?

Basically, there are three possible answers to this question:

1. Save the abstract form only
2. Save the concrete form only
3. Save both the concrete and the abstract form

Let's examine each one of these answers and decide which one to choose.

Saving the abstract form only seems to be sufficient, at least as far as the system is concerned, since it will be able to restore the file and reconstruct easily the original abstract tree without much work. The problem is that the only way the user can look at the concrete rules is by using the display facility. Therefore, he will not be able to have a clean and readable printing on which he can work and understand what is going on.

Saving the concrete form will satisfy the user's need, but in order for the system to restore the file and reconstruct the abstract trees, we will have to include a

parser. Thus, each time we make a restore, the system will have tc go through the parsing process to rediscover the structure and reconstruct the tree. Yet, any minor modification in the template, even when it does not affect the structure, will make the parsing impossible. For example: Suppose we had "fact n" as a part of a concrete rule, and suppose that the template used to construct this part was: "fact --".But, suppose for some reason (perhaps to improve the readability), the user had decided to change it so that now it looks like: "factorial --". Although this minor change does not affect the structure, the system will not be able to parse this same part of the rule.

On the other hand, using the abstract file the user can change the template as he likes as long as the structure remains the same. This is because, we don't record these key words in the abstract file, instead we record the structure. We don't even need the templates to restore the rules since the structure of the tree is preserved in the file.

The third answer suggests to save both forms. This solution seems to be the most appropriate. The user can have his printing whenever he needs, and the system can restore the rules in an easy and systematic way without any parsing required. In fact we will face similar alternatives when we discuss how the rules will be constructed. Figure 3.7 represents a printing of a saved rule file (i.e. abstract form), while figure 3.8 represents a printing of a its corresponding pretty_printed concrete forms.

. Commands

In order to distinguish between the two operations we will provide two different commands. Thus, to save the abstract form we use the command SAVE. On the other hand we will use the command PRINT to save the concrete form

```
rule1     01 tempeval 03 tempif 00 cond 00 ac 00 al
          03 tempevcon  01 tempeval 00 cond 00 ac 00 al

rule2     03 tempevcon 00 "true 00 ac 00 al 01 tempeval
          00 act

rule3     03 tempevcon 00 "false 00 ac 00 al
          01 tempeval 00 alt

rule 4    01 tempeval 02 tempfact 00 n
          01 tempeval 03 tempif 02 temequ 00 n 00 0
          00 1 02 tempmult 00 n 01 tempfact 02 tempsub
          00 n 00 1
```

Each node of the abstract tree is represented by a number followed by a string of characters. The number indicates how many dependents the subtree has, and the string represents the content of the node.

Figure 3.7 Printing of a Saved Rules File.

of the rules. Both commands can optionally include the name of the file. However, when the name of the file is not specified, the system will automatically assign the name of the list to the created file. As usual, an error message will be sent to the screen if there is no rule list created.

d. the restore function

This function allows us to load in the memory files which contain abstract rules. As we said earlier, loading the abstract file will be a relatively simple task with no parsing and no template search necessary. However, before starting the execution of this function, the system will check if there is already a rule list in the memory

```
rule1     : eval if cond then ac else al
            ==>
            evalcon cond  ac al
rule2     : evalcon "true ac al
            ==>
            eval act
rule3     : evalcon "false ac al
            ==>
            eval al
rule4     : eval fact n
            ==>
            eval if n = 0 then 1 else n * fact n - 1
```

Figure 3.8 Pretty_printing of a Rule File.

which have been created or modified but not yet saved. When such a list is found the system will interact with the user to get the confirmation to reinitialize the old list, or abandon the request. The new list will have the name specified in the command, or if no name is specified the system will assign to it the same name than the file.

5. Rule Manipulation

Operations on rules include insertion, deletion, displaying, search, and modification. Like in the template editor, and based on the same arguments, insertion may be done done during the editing mode, or independently. Moreover, all the operations on rules obey to the same mechanism described in the template editor using the same commands. The major difference, however, is that a rule is not created like a template. Therefore, we will focus more

on this aspect, and briefly discuss how the rest of the operations are designed.

a. inserting a new rule

As we mentioned several times through this chapter, a rule has two forms: (1) A concrete form which presents a readable and pretty_printed text constituted of key words, variable names, and constants. (2) An abstract form represented by a tree structure where we have only template names for the roots, and both variable names and constants for the leaves. The question now is how the user will input the rule and how the system will construct the abstract trees ?

Obviously, the classical approach is to let the user input the concrete rule, and leave it up to the system to build the abstract tree using the templates. Like we said earlier, this will require the system to scan the concrete rule, search for the templates which match the parts of the concrete rules, and subsequently construct the tree. Of course, if the user types incorrectly one of the key words, the system will nct be able to continue the parsing process. In this case it must issue an error message and either abandon the process or try to recover and continue the parsing, which may require some interaction with the user and perhaps a "dc what i mean feature". In sum we will have to deal with the same kind of problems encountered in conventional compilers.

Our objective is to take full advantage of the presence of the templates to optimize and simplify the tree construction, and eliminate a class of errors resulting from a misspelling in those key words. Yet, we want to get rid of them since, as far as the transformation is concerned, these key words have no semantic meaning. On the contrary, their presence will increase the amount of storage necessary for

the trees, and most important it will slow down the transformation, since there will be more nodes to compare for the matching and more nodes to copy during the substitution.

Another related problem, which has been introduced earlier (see RESTORE), is that the modification of key words in a given template will require the modification of all rules which include the templates. Otherwise, and since new programs will use the new template, these rules will never match a program subtree. On the other hand, using our method the user can change the key words he wants without affecting the rules and the transformation process. In fact, we can use this property to have different concrete forms for the same abstract tree. This may solve the problem of conflicting view points on how readability is perceived between users who share lists of rules and programs since each user can supply his own set of templates to unparse the same rules and have his own version of the concrete form.

In our solution, the user constructs a rule by puting together the parts of the tree using the basic subtrees whose structures are defined by the templates. He requests each template by its name. He will be assisted by the system which will display the text of the template with the place holders indexed by a number followed by a letter. The number represents the level of nesting which corresponds to the current height of the tree. The letter represents the position, from left to right, within the same height.

At the same time the system prepares a copy of the subtree, fills its root with the template name, and waits for the user to input the values of the son nodes. These input values may be a variable name, a constant, or a template name. In the latter case the node will expand to another subtree. The tree is constructed top down from left to right. Thus, at any moment, the user knows the current

position in the tree and what remains to input, simply by looking to the indices. For example:
Suppose we have already defined the following templates:

```
tempeval  : eval --
tempmember: member -- : --
tempnull  : null --
tempif    : if -- then -- else --
tempfirst : first --
temprest  : rest --
tempequal : -- = --
```

Each one of these templates defines a model of subtree by means of the place holders, and a concrete form by including key words. Now suppose we want to insert the following rule:

```
eval member x:l ==> eval if null l then "false
                         else if first l = x then "true
                         else member x : rest l
```

Using our solution the insertion will include the following steps:

- . step 1: input the rule name
- . step 2: input the analysis part
- . step 3: input the synthesis part

Figure A.1 illustrates the complete session for entering the analysis part of the above rule, with the different intermediate states of the abstract tree. Notice, for clearness, in each step we repeated the previous inputs and outputs, while actually a new input or output is just added. The arrow indicates the node to be filled, (i.e. the current position within the tree).

inputs and outputs	intermediate tree states
step 1 tempeval : < eval 1a > <1a>	tempeval -->
step 2 tempeval :< eval 1a > < 1a >tempmember:< member 2a : 2b > < 2a >	tempeval tempmember -->
step 3 tempeval :< eval 1a > <1a>tempmember: < member 2a : 2b > <2a>x	tempeval tempmember x -->
step 4 tempeval :< eval 1a > <1a>tempmember: < member 2a : 2b > <2a>x <2b>1	tempeval tempmember x y

When the tree is completed the system will automatically end the interaction and ask for the synthesis part which will be constructed using the same procedure.

Figure 3.9 Example of Rule Insertion.

The advantage of this approach is that it prevents the user from making syntactic errors resulting from a misspelling in the key words. It also allows the user to detect early an error in the spelling of the template name. For example, if he types "nul" instead of "null" the system will not find the template and subsequently doesn't display anything. Instead it will take the input as a variable name and will request the next input. Thus, the user will discover immediately that he made a mistake. because he did not get the template text as he expected. Or, if "nul" happens to be a template name, the user can see by looking at the displayed text that it is not the template he wanted. Another advantage is that the user can not enter an incomplete structure, because the system will keep asking him until the abstract tree is completed and will not accept any input after that.

Now the question is how the system is going to determine if the input value is a template name, a variable name, or a constant? To solve this problem, we will take the same convention adopted in most conventional programming languages. That is, template names are like reserved words, numeric constants are integer and real type numbers, nonnumeric literals are any string preceded by a double quote ("), and a variable name is anything else. Thus, once a name is assigned to a template it should not be used as variable name. Also, to avoid any ambiguity, a template name should not begin with a double quote or a digit. In fact if we recall, the system will issue an error message if a template is given an illegal name including the cases where the first character is a digit or a double quote.

b. displaying a rule

Most of our discussion up to now has referred to how the user will enter a rule and how the system will be

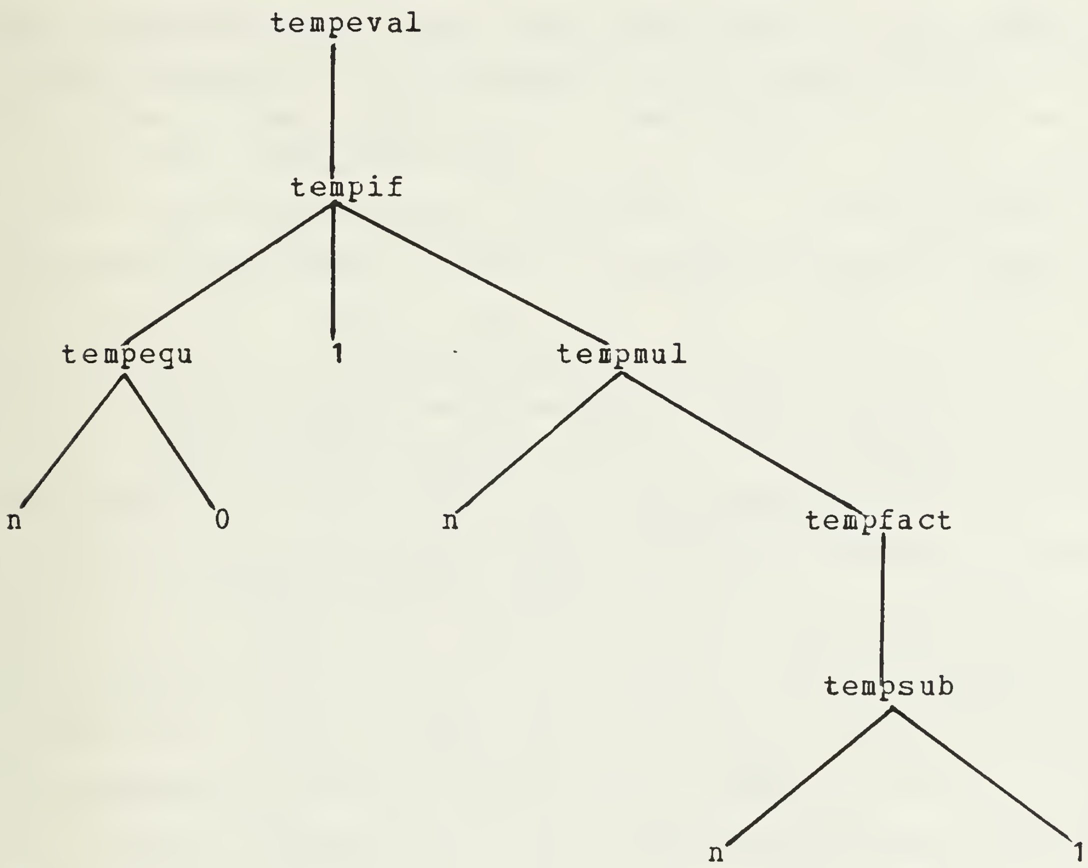
able to construct the abstract trees. However, for most users, the concrete form is the most visible aspect of the rules. It is the form which will be printed (see PRINT), and in the present case the form which will be displayed on the screen. But, since the abstract trees do not contain those key words which make a rule more readable by human beings, we must define a way to get back from the abstract form to the concrete form and sclve all the problems related to this process.

This process of unparsing the abstract rules will be relatively simple, provided the appropriate templates are supplied. It will consist for the system of walking through the tree, and for each new root searching for the template, embedding in the concrete form a copy of this template with its place holders filled with the son nodes. These nodes can be leaves representing a variable name or a constant, or another root representing a new templates to be embedded. Thus, the templates will be nested until the right_most leaf is reached.

Several problems may be encountered during the unparsing process such as the system not being able to find the template needed (i.e. no template with the name contained in the root exists), or the structure of the template not corresponding to what the system expects (i.e. number of place holders in the template is less or more than the number of sons in the subtree)

The first problem can be solved in two ways. The first solution would be to stcp the unparsing process as soon as a template is not found. The system would then display the unparsed part and send an error message with the name of the missing template. The second solution would be to display the name cf the missing template preceded by a special character and continue to unparse the rest of the tree.

This solution has the advantage of detecting all the missing templates in one pass. Because the templates are independent of each other, any missing one will not have an effect on unparsing the rest of the abstract tree. For example, suppose we have the following abstract tree:



now suppose we have only the following templates:

tempeval : evaluate (--)
tempif : if -- then -- else --
tempequ : -- = --
tempfact : fact --

Since the templates "tempsub" and "tempmul" are missing the system will display the following concrete form:

evaluate (if n = 1 then 1 else ?tempmul n fact ?tempsub n 1)

The second problem which can be encountered is that the structure defined by the template found does not correspond to the structure of the subtree the system needs to unparse (i.e. the template has been changed between the time the rule was entered and unparsed). In this situation we have two different cases. The first case is when the number of place holders is less than the number of sons in the abstract subtree. Conversely, the second case is when the number of place holders is greater than the number of sons.

The solution for the first case can be either to stop the unparsing process and send an error message or, the alternative would be, to continue and unparse the extra sons, but display them with a special note. For the same arguments mentioned above, we will chose the second alternative and display the extra scns between square brackets preceded by the name of the template. We added the name of the template because the extra sons or the place holders may appear far from its root in the concrete form, making it difficult for the user to determine to which template the extra son or place holder relate. For example:
If the template "tempif" was changed by "if -- then --", the same abstract tree will be unparsed and displayed as follows:

evaluate(if n = 1 then 1 [tempif n * fact n - 1])

This tells the user that "tempif" has now one less place holder than when used to construct the rule.

For the same reasons, the second case will be solved in a similar way. That is, the new extra place

holders in the templates will be displayed between brackets preceded by the name of the template. For example:
If the template "tempeval" has been changed so that it contains two place holders instead of one, the rule will then be displayed as:

evaluate (if n = 1 then 1 else fact n - 1 { tempeval -- })

This tells the user that "tempeval" has now one extra place holder than when used to construct the rule.

c. deleting a rule

This function allows one to delete the rule currently displayed. After the deletion the system will automatically display the next rule which becomes the current rule. However, when the end of list is reached the current rule will be "nil" and the message "end of list" will appear on the screen. In fact this function works exactly like the delete of the template editor.

d. searching for a rule

During the editing mode the user may search for a rule by typing its name. If a rule with the given name is found, it will be displayed and become the current one. However, if it is not found an error message will be sent to the screen and the system will return back to the situation before the search was requested.

e. modifying a rule

As in the template editor, the only way the rule editor provides to modify a rule is by deleting the old one and inserting in its place the new one. Of course, this is not a convenient way to do modification especially when a rule is long. Therefore it is preferable to include a more elaborate function which allows partial modification. But,

since we don't have a parser, the user has to work on the abstract tree. However, this task will be simple because all we can do is to replace a subtree or a node by another one. This can be done by a single command which specify the node or subtree to be replaced (using multiple qualifier to reference a node), and then the new node or subtree can be entered in the same way we enter a rule.

6. <u>Getting Information on the Current Rule</u>

This utility allows to get a message which tells the user what is the current rule. The command to get this information is: <u>CURRENT</u> Depending on what is the current position, the system will print one of the messages shown in table XIV.

TABLE XIV
Messages for the Current Command

1.	< a template name >
2.	"nil"; list is empty
3.	"nil"; end of rule list reached
4.	"nil"; no rule list has been opened or restored

7. <u>Exit the Rule Editor</u>

By typing the command <u>EXIT</u> the user will exit the rule editor,and return back to the user interface.

E. THE PROGRAM EDITOR

Recall that a program in our system has the same structure as a rule except that it forms a unique synthesis part. Also, like the rules, programs have an abstract form and a concrete form. It is therefore natural that we manipulate programs and their lists like we do for the rules. In fact we will use the rule editor to write and edit programs. However, in order to avoid the user geting confused we have made some adaptations so that he can make the separation clear. We have changed the prompt signal from (RE-->) to (PE-->) and we have replaced the word rule by program in all the messages and other system outputs.

1. Using the Program Editor

The program editor is given control either by the user interface when the command PGMEDIT is typed, or automatically by the interpreter when the user requested to save or print the result of the interpretation of his program. Once the program editor has started the user can request the same facilities available in the RULEDIT, using also the same commands to manipulate both programs or a program list. Thus, during one session we may have several programs grouped in the same list and ready to be interpreted without any loading and unloading being necessary.

2. Program Lists

There are two separate program lists which can be manipulated by the program editor. The first list is the one created by the user and which may contain any number of programs. The second list is created by the interpreter to hold temporarily the result of the interpretation, which might be either a entire tree, or a single value. The name of this list is "T.RESULT" and the name of the resulting

program is the same as the original program. However, since the program editor can not handle more than one list during a given session, it will not be possible for the user to access his program list when the program editor is given control by the interpreter. Conversely, the result list can not be accessed when the program editor is given the control by the user interface.

The question is: why is the program editor given the control by the interpreter? The reason is simply to allow the user to save, print, or redisplay the result (the result is automatically displayed after the interpretation). But, since this result can also be a new program tree, we thought it is natural to use the program editor facilities to perform these operations. Thus, the user can manipulate it like he manipulates any other list of programs. However, in principle, such things must be hidden from the user. That is, at the end of the interpretation or when the command to interpret is entered, the user specifies if he wants the result saved, printed or displayed. Whether the interpreter uses the program editor functions or its own functions must be irrelevant for the user and hidden from him, especially when this list is temporary and will be lost as soon as we exit the interpreter. Most of the users will probably ask why they can not access the result again since they just did it earlier using the program editor.

Now, you may have wondered why we make such a design decision since we think this should be hidden from the user and yet it may create some confusion. The main reason which motivates our decision is that we did not want to overload the interaction between the user and the interpreter with things which normally relate to program manipulation and not to its interpretation. For example, in order to be consistent we must allow the user to give a new name for the file to be saved, to specify if he wants to save the abstract or

the concrete form of the program, or both (which require two different file names), and perhaps allow him to delete some of the program results before saving or printing them. This, of course, will involve a lot of interactions, which have nothing to do with the interpretation and it would be inappropriate to incorporate them. On the other hand, the user will find it more natural to perform such operations while using the rule editor.

3. Exiting The Program Editor

To exit the program editor we will use the standard command EXIT. This will return the control back to the caller (i.e. either the user interface or the interpreter).

4. Limitations and Constraints

Since we are using the same features as the rule editor, we will have to deal with the same limitations and constraints concerning the modification of the programs. Yet, it may be worst because a program usually is longer and make take several lines to display. Therefore, it is necessary to provide a more elaborate way to modify programs other than by simply deleting the old one and inserting the new one.

Also, a program may be so big that it is not practical to display it entirely. Instead, it would be better if the user can ask to show only a given part, or truncate at a given point, with the possibility to navigate between the different nodes of the program tree.

In summary, we need a tool which makes uses of the structure of the program and the templates to perform the kind of facilities we described above. In fact it will be easy to identify and reference a node in the abstract trees by utilizing the template names used previously to construct the program.

F. THE INTERPRETER

This module is given the control by the user interface when the user types the command INTERPRET. Once the prompt signal (PI-->) is displayed the user can request the interpretation of one of the programs included in the program list. In addition, like the other modules, the interpreter offers a help facility, which may be obtained by typing the command HELP.

The user requests the interpretation of a program by typing its name. In addition, the user may ask the system to display the names of the rules successfully applied for the transformation process by typing the word "RULE" or simply "R" after the program name.

The interpretation process consists of the following steps:

1. Locating the Program List

In this step, the system will verify if the user has already loaded or created a program list. If such a list is found the system will go to the next step otherwise it will send an error message "can not interpret; no program list in the memory", and will redisplay the prompt signal. In this case the only operations allowed are either help or exit.

2. Locating the Program

Once the program list is located, the system will search for the given program until either it finds it or the end of list is reached. In the first case it will continue with the next step, in the second case it will issue the following error message:

"<program name> not found"

3. Creation of the Result List and Program Copy

In this step the system opens the result list called "T.RESULT" and copies the program to be interpreted into this list. This copy will be used for the transformation process. Thus, the original copy will be left unchanged after the transformation, so the user can request the interpretation of a given program as many times as he wants (perhaps with different sets of rules) without being forced to switch back to the program editor, and reload the program at each new interpretation.

4. Program Transformation

In this step we want to apply to the program the transformation rules until they no longer apply. When this occurs we will have the final state of the program, which may be either another program (i.e. an abstract tree), or a single value (i.e. a node which contains the final result). When the transformation is completed, the system will unparse the transformed program and display the concrete form as the final result.

The program transformation will be performed by a collection of functions which we will discuss below. Also, we will define the algorithms for each function, which are based on the analysis given in [Ref. 1]. These algorithms will be described using a Pascal like pseudo_language with comments included between brackets.

a. selecting and applying the rules

Basically there are two possible approachs to selecting the rules to apply for the program transformation process. The first approach consists of picking up a program subtree (the first subtree will be the program itself) and searching for a rule which matches this program subtree. If

such a rule is found the system will proceed to the synthesization of the program (i.e. perform the tree substitution). This process is repeated with the newly obtained program. On the other hand, if no rule matches the subtree, the system will pick up, in preorder, the next program subtree and restart the process with this new subtree. This process will continue until no subtree of the program matches any of the abstract rules.

The second approach consists of picking up the next rule in sequential order and matching it against the program subtrees. When a match occurs the system will proceed to the synthesization of the subtree and will restart the process with the resulting program. On the other hand, when no subtree of the program matchs the rule, the system will pick up the next rule from the rule list and start the same process again. Like in the first approach, the transformation will end when no rule matches any of the program subtrees.

In terms of number of comparisons as well as in term of implementation difficulty, both methods appear to be equivalent. There is no clear evidence about which method is more efficient. Therefore we believe that the best way to evaluate and compare them is by implementing each one of them and by having them tested over the same set of programs and the same set of rules. However, the second method has the advantage of maintaining the same order of selection which is the order in which the rules have been entered. This may be in some cases useful, and may reduce the number of rules needed to transform a program (e.g. recursive functions, where the basis must be checked before the recursion). Thus, the user can take advantage of this property when he writes his rules.On the other hand,in the first method the order of selection is random and hardly predictable especially in case of relatively large programs.

In our system we will use the second method because it preserves the order of the rules and,of course,it still works in the general cases. The general transformation algorithm is defined as follows:

```
ALGORITHM transform(program,rulelist);
   BEGIN
      end_of_transformation:=false;
      WHILE NOT end_of_transformation DO
      { transform the program by applying the rules
        until they no longer apply }
        BEGIN
          success := false;
          get_firstrule(r,rulelist);
          WHILE (NOT end_of_rulelist) AND (NOT success) DO
            { select the rules one by one until either a
              match occurs,or all the rules have been
              unsuccessfully tried for all program subtrees }
            begin
              get_firstsubtree(st,program);
              WHILE (NOT end_of_program) AND (NOT success) DO
                { search for a program subtree which matches
                  the rule until success or no more subtree }
                BEGIN
                  initialize(c);
                  match(r.analysis,st,c,success);
                  IF success THEN
                     synthesize(r.synthesis,st)
                  ELSE
                     get_preorder_nextsubtree(st,program);
                END WHILE;
              get_nextrule(r,rulelist);
            END WHILE;
          end_of_interpretation := NOT success;
        END WHILE;
END transform.
```

b. tree matching and variable binding

This process consists in matching the analysis part of the abstract rule against a given subtree of the program. The result of this matching process is either a failure, or a finite function whose domain is the set of all variable names in the abstract rule, and whose range is the set of values bound to these variables during the matching process. Thus, we need to define a procedure match(A,P,C,SUCCESS) where A is a pointer to the root of the abstract tree which initially will be the main root of the analysis, P is a pointer to the root of the program subtree to be matched against, C is the context of the variable binding, and SUCCESS is a Boolean variable which will indicate if the match succeeded or not. Note, since the values bound to the variable names can be either a single node which contains a constant, or a whole subtree, therefore the range of C will be a set of pointers to these single nodes or subtrees, and whose initial values must be "nil".

Using the same pseudo_language we define the algorithm for the match procedure as follows:

```
ALGORITHM match(a,p,c,success);
  BEGIN
    IF constant (content (node(a))) THEN
      BEGIN
         { case of constant to match }
         IF content (node (a)) = content (node (p)) then
            success := TRUE
         ELSE
            success := FALSE
       END  { end case of constant to match }
     ELSE IF is_template (content node(a)) THEN
       BEGIN
         { case of subtrees to match }
         { the content of the rcots must match and
```

```
      the rest of the subtree must also match }
    IF content (node (a)) = content (node (p)) THEN
      BEGIN
        { check the rest of the subtrees }
        s  := firstsubtree(a);
        s1 := firstsubtree(p);
        success := true;
        WHILE (s<>nil) AND (s1<>nil) AND (success) DO
          { match the rest of the rule subtree against
            the rest of the program subtree until
            either a failure occurs or no more subtree
            to be matched }
          BEGIN
            match(s,s1,c,success);
            { prepare next subtrees }
            s  := nextsubtree (a);
            s1 := nextsubtree (p);
          END WHILE;
        { at this point both subtrees must finish
           together otherwise match fails }
        IF s <> nil or s1 <> nil THEN
          success:=false;
      END
    ELSE
      { roots didn't match }
      success:=false;
  END { end case of subtree to match }
ELSE
  BEGIN
    { case of variable name in the node }
    IF in_domain (content node(a),c) THEN
      { p must match the value bound to the variable
        in the node pointed by a.We will use a special
        function called "equal" to verify the equality
        between trees patterns }
```

```
            success := equal (p, binding_of (content node(a))
          ELSE
            BEGIN
              { bind the variable to p and add it to c }
              success := true;
              include_bound (content node(a),p,c)
            END;
        END; { end case of variable }
END match.
```

c. synthesization (tree substitution)

The synthesization process will be done after the matching has succeeded and returned a finite function whose domain is the set of variable names found in the analysis part of the rule and whose range is a set of pointers which point to the nodes and subtrees bound to the variables. In addition, we have two pointers; the first one points to the program subtree which matched the rule, and the second pointer points to the synthesis part of the abstract rule. Thus, we have everything we need to start the synthesization process.

Basically, what the system will do is to take a copy of the tree representing the synthesis part of the rule, and then visit one by one the leaves of that tree (since a variable can not be found in a root node). If the leaf contains a variable name then the system will replace the leaf node by the value bound to the variable, which can be a whole subtree. On the other hand, if the leaf was a constant or a template name then it is left alone. Finally, when all the leaves have been treated, we detach the old program subtree from its father and attach the synthesized copy.

The algorithm which describes this function is defined as follows:

```
ALGORITHM synthesize (p,s,c);
  BEGIN
    copy (s,s1)
    get_first_leaf (leaf,s1)
    WHILE leaf <> NIL DO
      BEGIN
        IF is_variable (content (leaf)) THEN
          { extract from the range of c the pointer to the
            value bound to leaf, and make the replacement }
          BEGIN
            i := binding (content (leaf),c);
            detach (leaf,father(leaf));
            attach (i,father(leaf);
          END; { leaf treated }
        get_nextleaf (l,s1);
      END;
    { when all leaves treated }
    detach (p,father(p));
    attach (s1,father(p));
END synthesize.
```

5. <u>Displaying the Result and the Rules Applied</u>

In this step the interpreter will display the result of the transformation. Note that this result can be an entire program tree, in which case the system will unparse it using the same process we described in the rule editor and the program editor. But before the result is displayed,and only if the option 'rules' or 'r' was specified, the system will display the names of the rules successfully applied. Also, the total number of these rules will be automatically displayed at the end of the program interpretation.

6. Saving and Printing the Result

At the end of each program interpretation, the system will give the user the cpportunity to manipulate the result list. Thus, the system will ask the user the following question:

"Do you wish to access the result list yes/no: "

A "no" answer will cause the system to abandon the result list and redisplay the prompt signal (PI-->). A "yes" answer will cause the transfer of the control to the program editor where, as we explained in the previous chapter, the user can request any operation he needs cn the result list, such as save, print, or edit. In fact we did not impose any restriction on the allowable operations even though it does not make any sense to perform a restore, or an insertion on the result list.

When these operations are terminated, and as soon as the user exit the program editor, the control is return back to the interpreter. Note, at this point, the result is lost because the next time the user will have the opportunity to reaccess the result list will be after a new interpretation request, but the old result will be overwritten by the new one.

As an alternative, and since a result list can contain any number of program results, we could just append each new result at the end of the result list, without overwriting the old one. In fact this will not involve any change in the implementation, because the interpreter will use the insert function of the program editor, which takes care of multiple programs in the same list. However, this will create some practical problems. For example, suppose that during the interpretation session the user wants to save each result on a separate file. This will not be possible unless the user, after each interpretation,

switches to the program editor and deletes from the list the previous result, otherwise each new file will contain the old results in addition to the new one. Also, suppose that the user requests the interpretation of the same program more than once (perhaps using different sets of rules). In that case we have to decide, or let the user decide, whether to overwrite the old result, or create another one with the same name (since the result takes the name of the program), which may create confusion for the user.

We think that it is much simpler to treat the result list as a temporary list whose purpose is to serve as a copy for the transformation, and to give the user the possibility to save or print the result on a disk file.

7. Applying the Built_in Rules

Built_in rules are put on the top of the user rule lists. That means, they take precedence on the user's rules. These rules are ordered as listed in table XV When a subtree of the program matches the analysis part of a built_in rule (i.e. either a built_in template was used to construct the program subtree, or the subtree was previously transformed by applying a user rule), the interpreter will call the appropriate built_in function to execute the operation, and return a single node containing the result of the execution.

However, before the execution, these functions will check if the arguments are of compatible types (both numbers for arithmetic operations, and both numbers or both literals for boolean expressions). When an error is detected the function will return a subtree whose root is the name of the built_in template which corresponds to the appropriate error type, and whose two children are the arguments of the expression. These subtrees will later be unparsed according to the corresponding built_in templates listed in table X.

TABLE XV

Built_in Rules

```
*=============================================================*
*                                                             *
*                                                             *
*  rules for arithmetic operaticns                            *
*  -------------------------------                            *
*  number a  +  number b    ==>    sum a b                    *
*                                                             *
*  number a  -  number b    ==>    sub a b                    *
*                                                             *
*  number a /   number b    ==>    div a b                    *
*                                                             *
*  constant a * constant b  ==>  pro a b                      *
*                                                             *
*  sum,sub,div,and pro are built_in functions which will      *
*                                                             *
*  return  a  single  node  containing  the   result  of      *
*                                                             *
*  operation.                                                 *
*                                                             *
*                                                             *
*  Rules for boolean operations                               *
*  ----------------------------                               *
*                                                             *
*  constant a = constant b  ==> equ a b                       *
*                                                             *
*  constant a > constant b  ==> gre a b                       *
*                                                             *
*  constant a < constant b  ==> les a b                       *
*                                                             *
*  equ, gre, les are also built_in functions which will       *
*                                                             *
*  return a single node containing the value "true or         *
*                                                             *
*  "false.                                                    *
*                                                             *
*                                                             *
*  rule for eval                                              *
*  -------------                                              *
*                                                             *
*  eval constant a ==> a                                      *
*                                                             *
*  the built_in function evaluate will return a single        *
*                                                             *
*  node containing the value of the argument                  *
*                                                             *
*                                                             *
*=============================================================*
```

For example, giving the subtree "6 + 3.45.3" the system will return a subtree whose root contains "error2" and whose sons contain "6" and "3.45.3", which after unparsing will give:

'error: right argument is not numeric 6 3.45.3'

As an alternative, we could have decided to leave the subtree as it is (i.e. assume as the matching failed). In fact this would agree with the defination of the built_in rules, since the sons must be numbers for the arithmetic expressions, and of compatible type for Boolean expressions. Later, in the conclusion, we will explain why it would be better if we adopted the second alternative.

8. Exiting the Interpreter

As usual, we use the command EXIT to terminate the interpretation session,and return to the user interface.

IV. CONCLUSION

A. DESIGN ASPECTS

The purpose of this study has been the investigation and the implementation of a user friendly programming system based on tree transformations. Thus, in Chapter 2 we defined the various objects on which the system and the user will operate. Also, we defined and described a typical scenario of the different steps for developing programs, and we discussed the specifications of the programming environment. Along with this discussion, we defined a collection of tools which will support the user to accomplish all these steps. That is, create the templates and the rules and write, interpret, and debug the programs. In Chapter 3 we discussed the design and some of the implementation aspects, and, except for the debugger, we described the different facilities provided by each of the environment tools to manipulate the various objects.

The templates provided a means to define the semantic and syntactic framework for the language which will be used to write the rules and the programs, and will guide the system (via the place holders) to parse and unparse the abstract trees representing the rules and the programs. As we have seen, this parsing process is done efficiently with the minimum necessary storage, without scanning or token recognition except for determining if the input is a template name, a constant, or a variable name. Also, this process does not permit any syntactic errors, and prevents the construction of incomplete or incorrect structures.

The key words embedded in the template's text play the role of syntactic sugar to make the concrete form of the

rules and programs more readable, better formatted, and easier to understand by human beings. These concrete forms can be changed as the user desires by simply changing the key words, or by adding some more including comments. This will not have any effect on the rest of the processes (i.e. parsing, unparsing, and transformation). Hence, these changes can take place after the construction of the rules and the programs. This will make it possible to have many different concrete forms for the same program or rule, which represents a secondary application of the system to be used for the formatting of programs by supplying the appropriate set of templates.

Program formatting using our system, goes beyond the conventional formatting systems (i.e. indentation and spacing between lines of code), by involving the program text itself (i.e. the key words such as "if", "else", "=" etc.). This provides the user with a wide range of language levels which can go as high as his own natural language. However, as we know, this is only partially true because the user can not enter the concrete form as a normal text.

This raises the issue whether or not we should include a parser so the user can write the rules and the programs in the same way they are displayed. As you recall, in our present system, the user requests a template name and the system asks him to fill the place holders. In addition the system displays the text of the template; thus the user actually sees the concrete form with the place holders highlighted, which relieves him from the task of writing all these key words and making syntactic errors, which will increase the time necessary to write the concrete form, and the time necessary to construct the abstract trees. Instead, the user can give meaningful names to the templates so he can remember them easily (e.g. "if" for the if statement) during the insertion phase.

Adding a parser to the system would make it possible for the user to enter directly the concrete form using a conventional text editor, but it would in turn involve a lot of overhead since the system has to go through the process of scanning the concrete form, searching for the appropriate template (i.e. using pattern matching) to determine the structure it must construct. Also, this process is likely to generate syntactic and semantic errors (e.g. incorrect structure) and consequently it will affect considerably the efficiency of the system.

We think that our rule and program editors can be more flexible, more powerful, and more adequate in the present environment than a conventional text editor provided that we include an "undo" feature, and a more elaborate modification facility

1. The "Undo" Feature

The "undo" feature will allow the user to go backward in the construction of the abstract tree. That is, during the creation of a rule or a program, if the user discovers that he entered an incorrect input, he can request an "undo", which will cause the system to discard the value input with its corresponding tree structure, readjust the current position in the tree, and prompt again for the replacement input. For example, suppose the user wants to construct the following rule:

eval fact n ==> eval if n = 0 then 1 else n * fact n - 1

Further suppose that everything went correctly until the step to enter "n = 0 ", this means the intermediate shape of the abstract tree representing the above concrete rule will be as follows:

```
tempeval          ==>          tempeval
   |                              |
   |                              |
tempfact                       tempif
   |                           /  |  \
   |                          /   |   \
   n                     -->      |
```

As indicated by the arrow, the current position is then at the first son of "tempif" subtree. Now suppose the user typed "tempgreat" instead of "tempequal". As a result the system will display the text cf "tempgreat", create its corresponding subtree, fill its root with the name "tempgreat", set the current position to the first son of the created subtree, and wait for its value. By typing "undo" the user will cause the system to discard "tempgreat", destroy its corresponding subtree and readjust the current position such that it points to the upper level (i.e. in this case the first son of "tempif" subtree).

2. <u>The Modification Facility</u>

The modification facility has been already discussed in Chapter 3. It consists of making it possible to request the replacement of a subtree by another one. The old subtree is located by using, if necessary, multiple qualifiers. When it is found, the system will discard this subtree and ask the user to input the replacement using the same input method described for the insertion. For example, suppose we have already created the following rule:

eval fact n ==> eval if n = 0 then 1 else n + fact n - 1

Now we want to change "n + fact n - 1" by "n * fact n - 1". This would require to locate the subtree, which can be done by the command "locate synthesis.tempadd", where the qualifier "synthesis" is added just to speed up the search for

the subtree, since "tempadd" is sufficient to uniquely identify the subtree we need. When the node is located the system will display the subtree (i.e. in this case n + fact n - 1), and will ask for the replacement by displaying: <2c> (i.e. second level third son). At this moment, the user can start entering the new subtree in the same way as for insertion. However, to avoid retyping the same thing when there is no change, the user can press the return key to notify the system that the rest of the subtree remains unchanged. Also, if the new structure has more dependents the system will skip to the next input required for the extra independent. Of course, this may require some overhead due consistency checking between the new and the old structure, and may require facilities to do multiple replacements in the same subtree by jumping from one part to another.

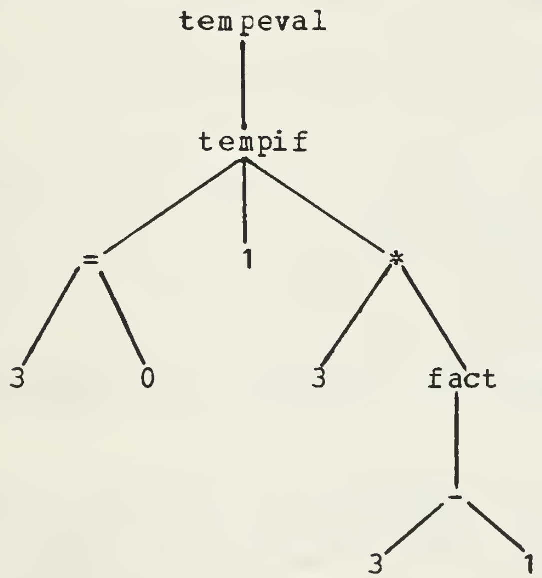
This raises the issue whether or not we should have included a structure editor which can accomplish such navigation operation, and allows the user to display and modify part of a rule or program, with the possibility of zooming in and out. After experimenting the system we felt that such editor with such facilities would be very helpful especially during the debugging process where, during the transformation, we need to lock at part of the program and the rules to find out the origin of the trouble (e.g. why a given rule didn't match a given part of the program, or what is the intermediate result after the application of a given rule etc.). However, a structure editor alone will not be able to do all these operations. Therefore, we need a debugger with the capability of answering specific questions such as:

1. Why a rule can not be applied to the program in general, or to a specific subtree?

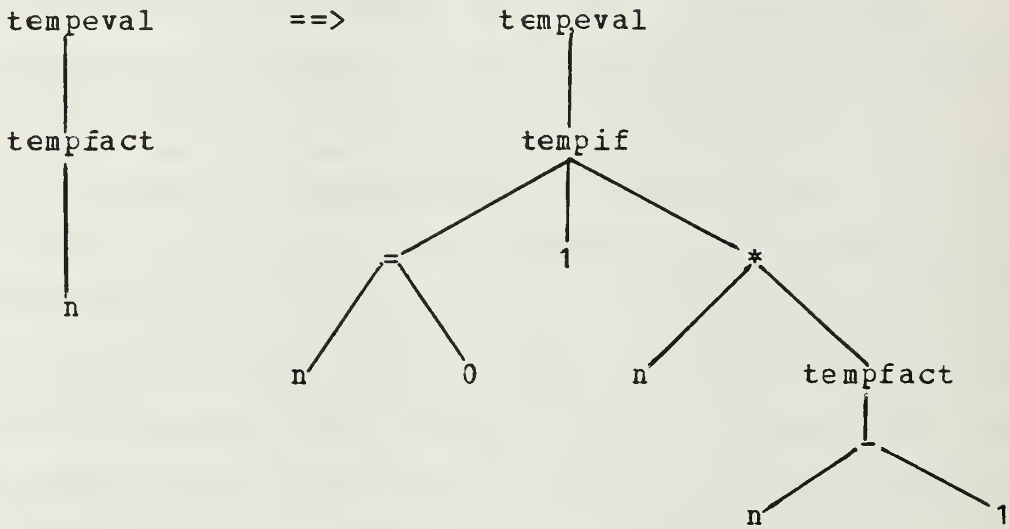
2. When a rule is applied and to which subtree of the program is it applied?

3. What is the intermediate result after a given rule is applied?

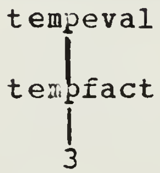
In addition, the user might be interested in performing the interpretation step by step with the intermediate result displayed after each step, the possibility of changing part of the program or the value of a given node to see how it would affect the transformation process and the result, altering the order of application of the rules by specifically giving the names and the orders of application, and finally being able to request backtracking. The latter request should not be difficult to implement, since it will take a switch which tells the system that now the synthesis part and the analysis part are inverted, and the order of the rules is also inverted (i.e. select the rules starting from the bottom of the list). With that, the interpretation should give a backtracking of the original one. For example having the following intermediate result:



This result is obtained by the application of the following abstract rule:



By inverting the rule and applying it the result will be:



The above abstract tree represents the previous state of the program (i.e. obtained by backtracking)

3. <u>Summary of the System Extension</u>

In summary we suggest the extension of the system in the following ways:

1. Overcome the problem of loosing control when an attempt to open a file fails because of an nonexisting file name for the RESTORE command. As we suggested in Chapter 3, this will be done by a special routine which must check the directory before an attempt tc open a file is made, and subsequently either go ahead and open it, or send an error message and return the control back to the user.

2. Change the template editor so that, like the other modules, only one list can be present in the memory at any given time. This will provide standardization of the modules, and avoid the user getting confused as happened during our experimentation with the template editor.

3. Restrict the operations the user can perform on the result list to SAVE, DISPLAY, and PRINT. This ensures, that the user can not do meaningless operations on the result list such as INSERT, RESTORE, and DELETE.

4. As you recall, we delayed the control of the lists which are created or changed during the session but not saved, until the user requests a "QUIT" (i.e. exists completly the system). Thus, although we suggested that it would be better practice to save the lists when switching from one module to another, we did not think that we should enforce it, or give a warning message because we don't know yet if the user will return back and make other changes. But, after we experimented with system we have decided that we should have included this control within each module because in some situations we lost control of the system without having the opportunity to go back to the appropriate editor and save those lists. One common situation which illustrates this inpredictable situation is when we the system went into an infinite loop during the interpretation process. In this case the only way to stop is to abort the job and return to UNIX.

5. Include a structure editor for the rules and the programs with the facilities we have described, including the "undo" and "modify" features.

6. Include a debugger which cooperates with the structure editor to provide the facilities and answer the kind of questions we have described.

B. SYSTEM EVALUATION AND USES

1. General Applications

So far we have been mostly discussing the design aspects of the programming environment. In the rest of this chapter we will focus on the programming system itself. That is, we will try to evaluate its advantages and its limitations. Also, we will make some suggestions about the possibility of extending its capability and improving its performance.

Our approach in evaluating "TTPS" (Tree Transformation Programming System) will be based on a comparison between "TTPS" and conventional programming systems. This analysis will be in terms of friendliness, appropriateness for a wider class of applications, and time required to produce a correct program to solve a given problem.

In "TTPS" friendliness is achieved in many aspects. The first aspect has been already discussed and concerns the facilities provided by the programming environment tools with the extensions we suggested.

The second aspect concerns the programming language. "TTPS" allows the users to have a wide variety of high level, formatted natural languages. Unlike with conventional programming systems, the user is not required to learn and master a formal language. Instead, he will define his own language framework reflecting his own perception of the syntactic and semantic aspects of his problem and adapted to his own style. This freedom has the advantage of eliminating the time required to learn and master a formal language before being able to use it. Also, it enables the user to tailor the language framework to be the minimum necessary to solve his problem since he will define only the templates and the rules he will need, thus eliminating the

classical problems which might result from the adaptaticn of the solution to the limited pcssibilities offered by the programming language. On the other hand, in "TTPS" we adapt the language to the solution, since we create it for the soluticn. Furthermore, as we have seen, the language can even be changed at any step of the programming development process. There is nc doubt this flexibility and power will have positive effects on the entire program life cycle including coding, debugging, and maintenance; and will result in a reduction of the time required to produce a correct program.

The third aspect concerns the programming style. With "TTPS" the user is liberated from being tied to a unique programming style. Instead, it allows him to define the style he wants, which can be a functional type, a conventional type, a combination of both, or his own specific style.

The templates and the rules represent an elegant, easy, and natural way to define the syntax and the semantic behavior of the programming system. With the rules we can simulate every kind of behavior including those of the constructs defined in conventional languages such as while, for, case, and if constructs. All this is done using one type of rule which maps trees into trees, thus providing a uniform, clear, explicit, and natural way of reasoning.

These properties make it simple for the user to reason about the program behavior at a high conceptual level, since with trees he can define very large computational units, which can be processed in parallel. However, when the user desires, these trees can still represent small units such as assignment statements, or memory allocation. By using larger units, programs can be constructed faster. Also, the debugging process will be easier since with these large units, and with the assistance of the system, the user can quickly locate the origin of the bug.

In addition "TTPS" provides an appropriate environment for developing functional programs. As explained in [Ref. 1], functional programming is important because it encourages one to think at a higher level of abstraction, it provides through its large units (i.e. trees in our system) a methcd for programming large, parallel computers, and it provides an adequate framework for AI applications.

We believe that the majcr difference between conventional programming systems and "TTPS" is that the first category provides primitive operations and the user has to use them to construct his program and define its semantic behavior through more complex structures (e.g. procedures, and functions), with all the related problems of parameter passing and side effects. In conventional programming systems we have two separate entities; the programming language and the program which represents a specific instance which may or may not fit well in the language framework. Therefore the user has the added burden of adapting the problem solution to the features available in the language. On the other hand, in TTPS these two entities are inseparable because the language framework is designed to solve the problem; therefore the user does not have to deal with external constraints. Also, the rules are independent from each other, thus the user does not have to worry about side effects, and feature interaction. These properties makes "TTPS" appropriate for a wider class of applications.

2. Other Specific Applications

Along with this classes of functional and conventional applications, "TTPS" can be used for other specific applications such as:

a. simulation of a Syntax Directed Editor

Using the templates we can define the grammar for the legal syntax, and then using the program editor we can construct the program tree by requesting the appropriate template for each type of construct. When unparsed and printed this abstract tree will give the text of the program, which then can be used normally as a program text. For example, the template which describes the structure of a Pascal "for loop" would be written as:

```
tempfor   : for -- := -- to --
                --
```

b. formatting programs

This application has been already discussed; it consists of using the templates to describe how the program is to be formatted. This include indentation, spacing between lines and words, comment insertion, and word substitution. However, without a parser only programs written using the "TTPS" program editor can be formated. Thus, in order to be general we need to add a parser capable of transforming a unformatted program text into abstract tree, and then by supplying the appropriate set of templates we can have the same program unparsed and printed in its new form.

c. structure transformation, and string translation

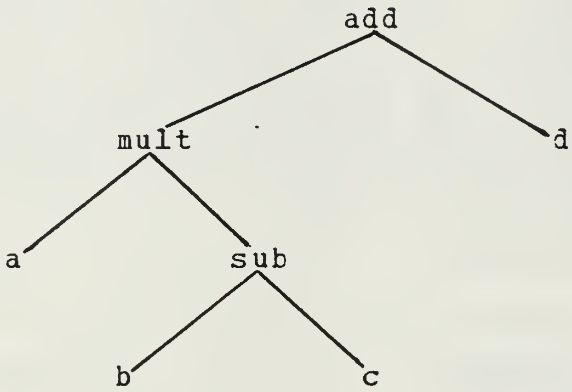
Given with the appropriate templates and if necessary the rules, the system is able to transform any program structure to another one, including string translations. For example, suppose we have the following set of templates:

```
add       : (-- + --)
sub       : (-- - --)
mult      : (-- * --)
```

Using these templates we can write the following expression:

((a * (b - c)) + d)

The abstract tree which represents this expression is as follows:



Now, by replacing the old templates by the following ones:

add : -- -- +

sub : -- -- -

mult : -- -- *

when unparsed, the same tree will be printed as:

a b c - * d +

Observe that, this translation does not require any rule.

As another example of a structure transformation which requires the application of transformation rules, we can transform a Pascal case construct into a sequence of PL/1 if statements. For instance, if we have the following case construct:

```
CASE country of
     1: WRITELN ('country is Tunisia');
     2: WRITELN ('country is France');
     3: WRITELN ('country is Greece');
     4: WRITELN ('country is USA')
END; (* case of *)
```

After transformation we will obtain the following sequence of PL/1 if statements:

```
IF country = 1 THEN
   PUT SKIP LIST ('country is Tunisia');
ELSE IF country = 2 THEN
   PUT SKIP LIST ('country is France');
ELSE IF COUNTRY = 3 THEN
   PUT SKIP LIST ('country is Greece');
ELSE IF COUNTRY = 4 THEN
   PUT SKIP LIST ('country is USA');
```

This transformation is done using the templates listed in Figure 4.1, the concrete rules shown in Figures 4.2 and their corresponding abstract forms shown in Figure 4.3. Note that the source structure can be a program written in a user defined form which can be transformed to a given language before being compiled.

d. code generation

In this application we use the rules to generate the code. That is, the analysis part of the rule represents the source program tree, and the synthesis part represents the instructions which will be generated when a match occurs between a program subtree and the analysis part of a rule.

e. system programming.

In this application we use the rules to transform a given tree structure into a system call, which then can be executed to return whatever result, which will replace the original subtree. For example:

read file m ==> call i/o_routine1 input_value

This rule, when applied, will causes the execution of "i/o_routine1", and the replacement of the subtree "read file m" by the input_value.

```
case        : CASE -- OF
                     --: --;
                     --
-----------------------------------------------
nextcase    :        --: --;
                     --
-----------------------------------------------
lastcase    :        --:--
              END; (* case of *)
-----------------------------------------------
pl1_if      : IF -- THEN
                 --;
              ELSE --
-----------------------------------------------
pl1_lastif: IF __ THEN
                 --;
-----------------------------------------------
writeln     : WRITELN (--)
-----------------------------------------------
put         : PUT SKIP LIST (--)
-----------------------------------------------
equal       : -- = --
```

Figure 4.1 The Templates for Case Transformation.

3. Limitations and Constraints

a. Constants, and variable naming

In "TTPS" the length of constants and variable names is limited by the size of the data we can store in a tree node, which in the present case is limited to 20 characters. Thus, in order to be able to handle strings longer than this number, the user must define a structure which he can call "concatenation". This structure concatenates strings of 20 characters to obtain larger ones.

```
firstcase : CASE v OF
                 m: x;
                 w
            ==>
            IF v = m THEN
               x;
            ELSE w
-------------------------------------------------------
midlecase : IF v = m THEN
               x;
            ELSE n: y;
                 w
            ==>
            IF v = m THEN
               x;
            ELSE IF v = n THEN
               y;
            ELSE W
-------------------------------------------------------
lastcase : IF v = m THEN
              x;
           ELSE n: y
           end;
           ==>
           IF v = m THEN
              x;
           ELSE IF v = n THEN
              y;
-------------------------------------------------------
write_r  : WRITELN (s)
           ==>
           PUT SKIP LIST (s)
```

Figure 4.2 Concrete Transformation Rules for Case.

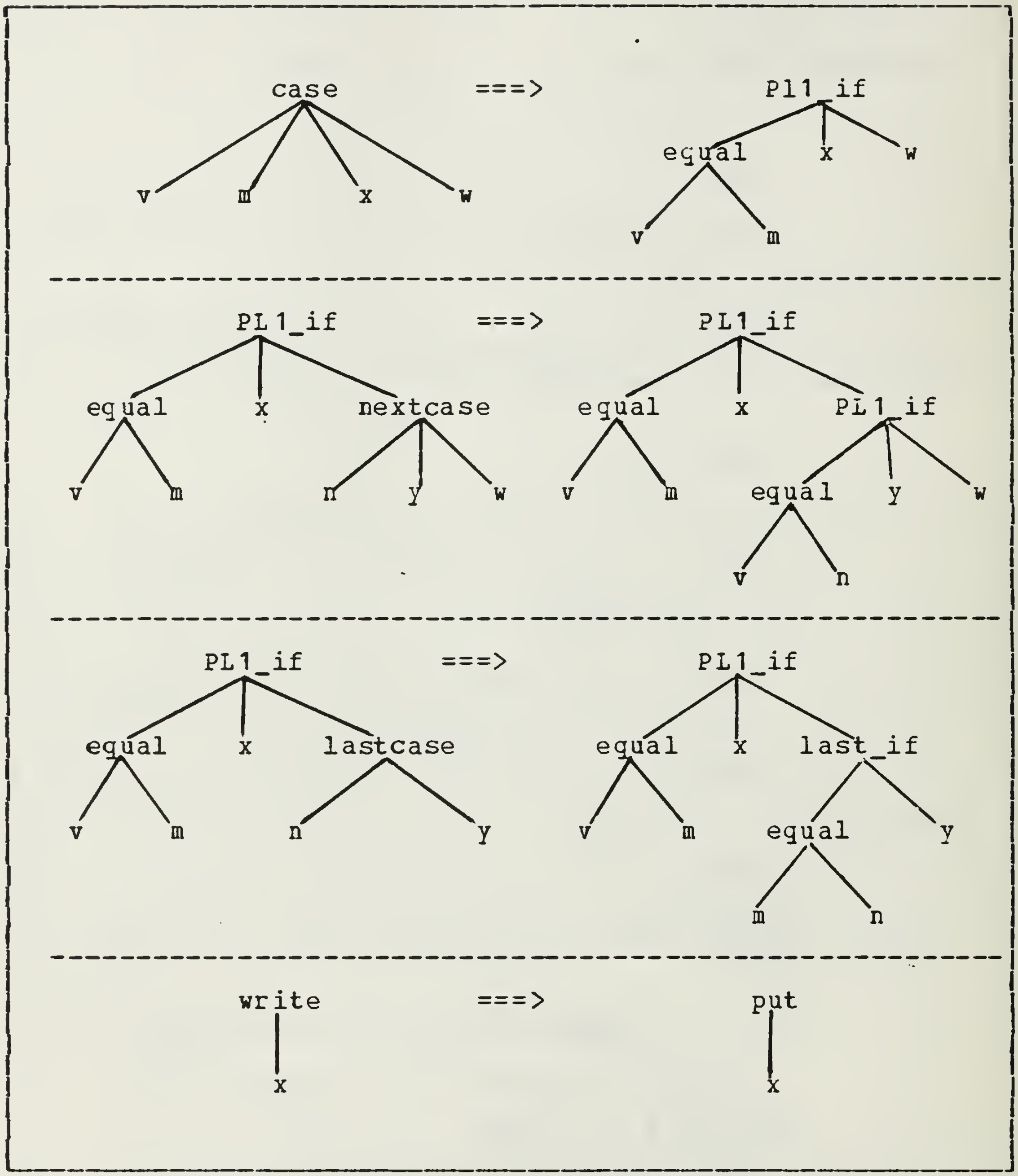


Figure 4.3 Abstract Transformation Rules for Case.

An alternative solution would be to define a template for each string larger than 20 characters, and then use the template name, in the rules and the programs, to reference these strings.

Both solutions seem to lack flexibility and generality, since they will not work for the variable names and numbers larger than 20. Also strings will be stored in 20 characters long nodes, thus resulting in waste space due to internal fragmentation, and in an increasing complexity in the rules which handle these strings. For all these reasons we suggest the implementation of a more general solution, which consists of representing the data stored in the tree nodes by linked lists. Thus, the content of each node is variable rather than fixed like it is presently implemented.

b. built_in rules

In order to improve the performance of the system, we think it is necessary to include more built_in rules, such as rules for mathematical functions (e.g. square root, exponentiation etc.), rules for list manipulation (e.g. first, rest, null etc.), and rules for input/output functions (e.g. open, read, write, display). In fact these rules are now being implemented. The open rule (1) takes a subtree whose root contain the name of the "open template", whose left son contains either "input" for input file or "output" for output file, and whose right son contains the name of the file to open, (2) opens the file, (3) and disable the request by replacing "open" with "opened". The "read" rule takes a subtree whose root is "read template name" (i.e. "read") and whose left and right sons contain respectively the name of the file, and a variable name, (2) replaces it by the input which can be a constant value or whole subtree. The "write" rule (1) takes a subtree whose root contains "write" (i.e. the name of the write template), and whose left and right sons contain respectively, the name of the output file, and the value to be written, (2) displays it, and (3) disables the write request by changing

the content of the root to "Written". Thus, the user can reactivate it when he needs to, by inserting at the appropriate place in the rule list a rule which looks like: "written x m ==> write x m"

The "display" rule is similar to the "write" rule but its second son may be a whole subtree which must be unparsed and displayed; then the request is disabled by replacing "display" with "displayed".

APPENDIX A

USER'S GUIDE TO TTPS (A TREE TRANSFORMATION PROGRAMMING SYSTEM)

A. INTRODUCTION

"TTPS" is a programming environment which includes four integrated tools: A template editor, a rule editor, a program editor, and an interpreter.

On the top of these modules there is the user interface whose function is to allow the user to log on and off, initialize and control the session, and perform the switching between the modules.

B. TYPING THE COMMANDS

A "TTPS" command may be typed entirely on one line, or broken into a sequence of subcommands. Thus, if you type only the name of the command, the system will keep asking for the remaining information until it is able to execute the command properly, or until an error was detected. However, if during this interaction you wish to abandon the request you may do it by pressing the escape key in place of the next input.

Example:

Suppose you started a load operation by typing RESTORE, the system will ask you for the name of the file, but for some reason (perhaps you forgot the name of the file and you want to check the directory), then instead of inputing the name press the escape key. The system will automatically disregard the command.

All the commands and their arguments can be typed using either upper case or lower case letters or a combination of both.

C. STARTING THE SESSION (TTPS)

When you type TTPS the system will start the session and give you the prompt (-->) to select the module you want to run or to request help.

D. GETTING HELP (HELP)

By typing HELP the system will display on the screen the informations relative to the module currently running.

E. SELECTING A MODULE

TEMPEDIT : selects the template editor.
RULEDIT : selects the rule editor.
PGMEDIT : select the program editor.
INTERPRET : select the interpreter.

F. ENDING THE SESSION (QUIT)

To terminate the session and exit the TTPS you type QUIT. However, before the system logs you off it will check if there is any list left in the memory which has been created or modified during the session but not saved. If such a list or lists are found they will be listed on the screen, and you will be asked if you want to save these lists or drop them. When you answer by "NO" you confirm the "QUIT", and the list will be destroyed. On the other hand, if you answer "YES" you will get back the control, but it is still your responsibility to save the lists you want to by selecting the appropriate module and command.

G. USING THE TEMPLATE EDITOR

The template editor offers a collection of facilities to manipulate both the templates and the template lists. But before we present the different commands, let's make clear the notion of current list and current template.

The template editor allows many lists to be simultaneously present in the memory which can be manipulated alternatively one at a time. Therefore, the current list is the latest one which have been involved with a list operation, except for remove where the next list becomes the current one, or "nil" when there is no next list to the removed one. These same rules apply to the notion of current template within a list, but the current template will take the value "nil" at the end of list or when the list is empty.

1. Built_in Template

These templates are part of the system. They are grouped in one list named BUILT_TEMP loaded automatically at the starting of each session. Once loaded the list can be accessed like any other template list.

2. Open a List (OPEN LISTNAME)

This command allows one to create and initialize a new list, or locate an existing list. In both cases the specified list becomes the current list, but this will not allow one to start the editing yet. This involves the next command.

3. Edit a List (EDIT LISTNAME STARTING POINT)

This command permits to start the editing session on the given listname, provided that the list have been already created by an OPEN or RESTORE command. The listname may be omitted in which case the system will assume by default the current list.

The starting point will indicate the place from where the editing will begin. It can be one of the following points.

FIRST : start from the beginning of the list

LAST : start at the end of the list

TEMPLATENAME : start at the given template

NOT SPECIFIED: start from the current template

The editing session will begin by displaying the template which corresponds to the given starting point.

a. display the next template (RETURN KEY)

By pressing the return key you will get the display of the next template or a message indicating that the end of list is reached.

b. insert a new template (INSERT)

The new templates will be inserted before the current template. However, if the list is empty the system will automatically start the insertion at the beginning of the list. Insertion may be done at the end of the list when the message "end of list reached" appears on the screen.

Each new template must have a name and a body. Thus, the system will give the prompt to type the name which may include from one up to ten characters. Next, it will ask you to start typing the body of the template which is a combination of words and place holders signified by two successive minus signs (-). The template text is ended by the escape key. The end of the insertion is also indicated by the escape key entered in the place of the name.

c. deleting a template (DELETE)

This command allows the deletion of the template currently displayed. After the deletion, the next template will be automatically displayed and then becomes the current

one. Eventually the message "end of list" will appear when the template deleted happens to be the last one in the list.

d. search (TEMPLATE NAME)

During the editing of a list, it is possible to search for a given template by typing its name. When found it will be displayed and subsequently become the current one. On the other hand, if no template with the given name was found, the system will send an error message and return back to the previous situation.

e. ending the editing session (ESCAPE)

By pressing the escape key you exit from the edit mode, but the control remain in the template editor.

4. <u>Direct Insertion</u> (<u>INSERT PLACE OF INSERTION</u>)

This is an additional facilities which allows direct insertion without being in the editing mode. It can be used when you need to make an insertion only, and at a particular place of the list. The place can be:

FIRST : insertion at the beginning of the list
LAST : insertion at the end of the list
TEMPLATE NAME: before the given template
NOT SPECIFIED: before the current template of that list

5. <u>Saving a List</u> (<u>SAVE FILENAME LISTNAME</u>)

This command have the effect to save the given list on a disk file and assign to that file the given name.Both names can be omitted in which case the function will save the current list and assign its name to the created file.

6. <u>Restoring a List</u> (<u>RESTORE FILENAME LISTNAME</u>)

This command loads the specified file with the given listname. When no listname is given the restore function

will automatically assign to the list the name of the file. However, you should be aware of two important things concerning what happens when the given file does not exist and when there is already a list in the memory with the given name:

. Case 1

When no such file is found, the system will abort the job and subsequently the content of the memory will be lost. Therefore it is better practice to save things which are created or modified during the session, before you issue a restore command unless you are sure you have given the correct file name. For this reason the system will send a warning reminding you of this fact, and will give you a second chance to verify the input file name, and in consequence either correct it and confirm the request, or abort it for more verifications.

. case 2

When the given listname is already assigned to an existing list, the system will automatically ask you to confirm the overwrite of the old list or give another name or simply abandon the request by pressing the escape key.

7. Removing a List (REMOVE LISTNAME)

This command allows to remove a list from the memory and free the occupied space. However, when the list to be removed was created or modified during the same session but not saved, the system will send a warning message and give you the opportunity to abandon the request and save the list or confirm it.

Like the other commands,if the listname had not been specified, the system will take the current list as default value.

8. <u>Merging two Lists</u> (<u>MERGE LISTNAME1 + LISTNAME2 = LISTNAME3</u>)

This command allows the concatenation of the second list with the first list yielding a third list. as usual, the current list will be taken as default value for any one of the three lists if they are not specified. Thus, at the limit it is possible to concatenate a list with itself. Listname3 can be either a new list or an existing one. The latter case will be treated like an overwrite.

Note that here we have a situation where you may have duplicate templates in the same list. When that happens the system will notify you by giving the names of all the duplicates found, but it is your responsibility to delete the one you don't need or give different names.

9. <u>Listing the Template Lists</u> (<u>LIST</u>)

When you type this command the system will display a listing of the names of all lists currently in the memory, and the number of templates each list contains. "nil" will be given when there is no list in the memory.

10. <u>Inquire About the Current List and Template</u> (<u>CURRENT</u>)

By typing this command you will get a message indicating the name of the current list and the current template within this list.

H. USING THE RULE EDITOR (RULEDIT)

Like the template editor, the rule editor provides facilities to manipulate individual templates and lists of rules as a whole entity.However,now we can have only one list in the memory. Thus, we will not have the notion of current list, but we still have to keep track about the current rule.

In general, we manipulate lists of rules in the same way we do with lists of templates. There are, though, some differences resulting from the fact that we have only one list. Also, now we have to deal with two different forms of the rules: the concrete form and the abstract form.

In the rest of this section we will introduce the different commands to use the facilities provided by the rule editor. However, we will simply mention the ones that have been described in the template editor and spend more time with the description of the new commands.

1. Built_in Rules

Like built_in templates, built_in rules are an integral part of the system which can be applied to the program like the other rules. The synthesis part of these rule can be either a single constant value or an error type.

Built_in rules can not be accessed by the rule editor, therefore we will mention them here.

a. rules for arithmetic operations

These rules take two operands, verify if they are numbers and produce the result. The list includes rules for the addition (+), the multiplication (*), the subtraction (-), and the division (/).

b. rules for Boolean operation

These rules take two operands, verify if they are of the same type (i.e both numbers, or both literals), and produce a Boolean value ("true or "false). The list includes rules for equality (=), greater than (>), and less than (<).

c. evaluate rule

This rule will take its unique argument and evaluate it to return a single value which may be either a number, or a nonnumeric literal. Note that numbers include integers and reals. Numbers are written as they are. On the other hand nonnumeric literals must be preceded by a double quote (").

2. Open a Rule List (OPEN LISTNAME)

This command will create a new list, or if the list exists, it will reintialize it. The second case may require the confirmation of the request if the old list has been changed during the session but not saved yet.

3. Editing a Rule List (EDIT STARTING PLACE)

This function works exactly like the one described in the template editor. It provides the same operations (INSERT, DELETE, DISPLAY, AND SEARCH). Of course, writing a rule is different than writing a template.

Each rule comprises three parts:The name of the rule,the analysis part, and the synthesis part. The name is entered first and may include from 1 up to 10 printable characters. When finished the system will ask you to enter the analysis part, and then the synthesis part. Both parts are constructed using the basic tree structure defined by the templates.

The best way to understand how a rule is entered is to go through an example. However, It will help if you think in terms of tree structure rather than in terms of concrete structure, because when you write the rule you are in fact building the tree at the same time, or more precisly the system prepares the tree and you fill the nodes with constants, variables, or template names which will cause the creation of new subtree.

Example: Suppose we have already defined the following templates:

evaltemp : eval --

iftemp : if -- then -- else --

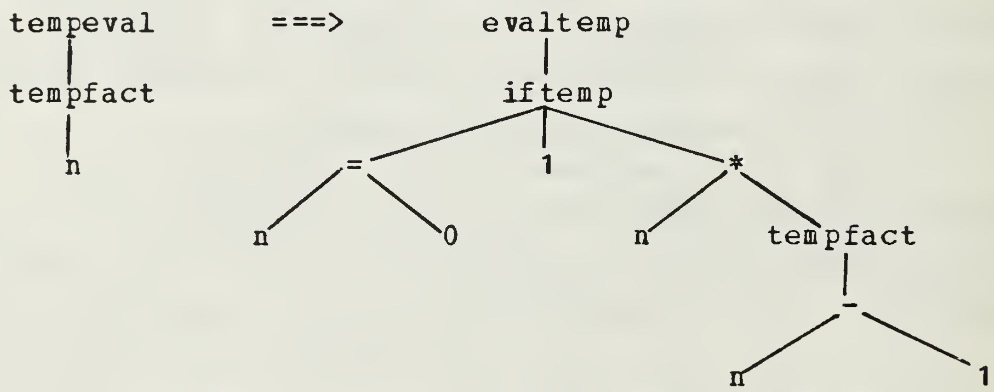
factemp : fact --

In terms of the tree the, first template tells us that an eval subtree will have one son,the second template tells us that an if subtree will have three sons,and the last template tells that a fact (factorial) subtree will have one son.

Now suppose we want to write the following rule:

eval fact n ==> eval if n=0 then 1 else n * fact n - 1

the abstract tree which corresponds to the above concrete rule will look like



Now, how do we enter the rule so the system can built the above tree? Since we don't have a parser to discover the structure from the concrete form, we will use the templates to tell the system which structure we need.Templates are requested by their name.

Figure 3.9 represents the entire process for entering the above rule. The inputs are written in capital letters while the system output are written in small letter.

```
rule name: FACTRULE
analysis part
-------------
EVALTEMP  < eval  1a >
  <1a> FACTEMP < fact 2a >
         <2a> N
synthesis part
--------------
EVALtemp  < eval  1a >
   <1a> IFTEMP    < if 2a then 2b else 2 c >
          <2a> =          < 3a = 3b >
                  <3a> N
                  <3b> 0
         <2b> 1
         <2c>*          < 3a * 3b >
               <3a>N
               <3b> tempfact  < fact 4a >
                      <4a> -        < 5a - 5b >
                             <5a> n
                             <5b> 1
rule inserted
```

Figure A.1   Example fcr Inserting a Rule.

Notice that when we request a template the system will automatically display the format of the template with the place holders indexed. The number in the index indicates the level of nesting which corresponds to the height of the tree. On the other hand, the letter indicates the position, from left to right, within the template which correspond to the position of the son node of the subtree.

The last remark concerns the templates. As you noticed, we did not need to define the templates =, *, and -, because we used the built_in templates.

4. Direct Insertion (INSERT PLACE OF INSERTION)

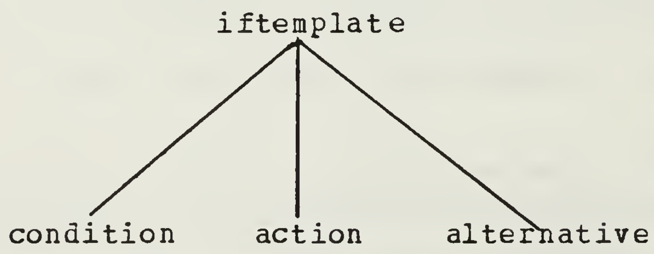
This command allows direct insertion without being in the edit mode. Basicly it works like the insertion from the edit mode, except now you can specify explicitly the place where to start the insertion. The place of insertion can be first, last, a template name, or if omitted, the insertion will be before the current rule.

5. Saving a List (SAVE FILENAME)

This command allows one to save the abstract form of the rules. When the filename is not specified the system will assign the name of the list to the created file.

6. Printing a List on a Disk File (PRINT FILENAME)

This function will send a pretty printing of the unparsed rule to a disk file. Basically it works like the save function except, as we said, each rule is unparsed according to the format given by the templates. For example, having the following tree as part of an abstract rule:



with the following template:

```
iftemplate: if -- then
                 --
               else
                 --
```

the abstract tree will give the following concrete form:

```
if condition then
   action
else
   alternative
```

However, if the system can not find the appropriate template, in our present case "iftemplate", he will simply print in preorder the subtree preceded by a question mark. Thus, the above subtree would be printed as:

```
? iftemplate condition action alternative
```

7. Restoring a List (RESTORE FILENAME LISTNAME)

This command allows one to load into the memory files containing only abstract rules. Thus, concrete rules cannot be loaded because the system will not be able to parse them. An error will occur when an attempt is made to restore files which dc not have the adequate structure.

For the rest of the details on this command refer to RESTORE in the template editor.

8. Inquire About the Current Rule (CURRENT)

When you type this command the system will give the name of current rule, or "nil" when at the end of list.

9. Ending the Rule Editor (EXIT)

When you type exit you terminate the template editor and return back to the user interface where you can select another module or simply quit the system.

I. USING THE PROGRAM EDITOR

In "TTPS" a program is treated like a rule except that programs include only one part. Thus, we will use the same facilities provided in the rule editor with rule list becoming program list, and with the rules replaced by programs. Thus, it is possible to have several programs in the same list, each one treated by the program editor like a rule is treated by the rule editor.

For more information about how to use the different facilities refer to the rule editor section.

J. USING THE INTERPRETER (INTERPRET)

As we said earlier, this module is run by the command INTERPRET. Once it takes control it will give you the prompt to interpret your programs. The interpretation is started by typing the name of the program, which may be followed by the option RULES or simply R. When this option is specified the system will display in sequential order the rules applied to the program. At the end it will give the number of rules successfully applied to your program.

When the interpretation is completed you will have the opportunity to save or print the result list called "T.RESULT" on a disk file by an automatic switch from the interpreter to the program editor. You may decline the offer by simply giving NO as an answer to the question displayed at the end of the interpretation.

Notice, that after you save or print the result the control will return back to the interpreter so you can request the interpretation of the same program or an other one from the program list.

1. Exit the Interpreter (EXIT)

Like the other modules you exit the interpreter by the command EXIT. This will give the control back to the user interface.

LIST OF REFERENCES

1. MacLennan, Bruce J. *Functional Programming Methodology: Theory and Practice*. To be published by Addison_Wesley.

BIBLIOGRAPHY

Aho, A. V., Johnson S. C., "Optimal Code Generation for Expression Trees" JACM 23, 1976.

Center for Research in Computing Technology, Havard U., Cambridge, Mass., Technical Report 9 - 73, Tree Transductions and Families of Tree Languages, by B. Baker, 1973

Backus, J., "Can Programming be Liberated from the Von Newmann Style? A Functional Style and its Algebra of Programs" Communications of the ACM, volume 21 number 8, August 1978

Barrett, W. A., Compiler Construction: Theory and Practice. Science Reasearch Associates, 1979

Computing Laboratory, U. of Newcastle Upon Tyne, England, Technical Report 50, Transformational Grammars for Languages and Compilers, by F. DeRemer, 1973

Killy, J. F., "An Interactive Design Methodology for User Friendly Natural Language" ACM Transactions, March 1984.

Kron, H. H., Practical Subtree Transformational Grammars. Master Thesis, U.C. Santa Cruz, California, 1974

Kron, H. H., Tree Templates and Subtree Transformational Grammars. PH.D. Dissertation, U.C. Santa Cruz, California, 1975

Waters, R. C., "The Programmer's Apprentice:Knowledge Based Program Editing" IEEE Transactions, January 1982

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California93943	2
4.	Computer Technology Curricular Office code 37 Naval Postgraduate School Monterey, California 93943	1
5.	Professor Bruce J. MacLennan, Code 52 ml Department of Computer Science Monterey, California 93943	1
6.	Professor Gordon H. Bradley, Code 52 bz Department of Computer Science Monterey, California 93943	1
7.	Ministere de la defense nationale Direction du Personnel et de la Formation 1, bd Bab M'Nara Tunis Tunisia	1
8.	Capitaine Chok M. Bechir Ministere de la Defense Nationale DPPI 1, Bd Bab M'Nara Tunis Tunisia	1
9.	Lt.col Abdoulaye Dieng Etat Major des Armees BP 4042 Dakar Republic of Senegal	1
10.	Major Harilaos Papadopoulos Hellenic Air Force General Staff Agia Barbara - Aigaleo Athens - Greece	1
11.	Captain Nasser Alsubaiei SMC 1096 NPgS Monterey, California, 93943	1

211179

Thesis
C448864 Chok
c.1 Investigation and
implementation of a
tree transformation
system for user
friendly programming.